API设计与实现

主讲人：陈长兵

目 录

CONTENTS

4 API设计模式

- 标准方法
- 定制方式
- 部分更新与检索
- 批操作
- 分页
- 长耗时操作
- 可重复运行的作业
- 导入导出

4.1标准方法-背景

- 如何快速学习和应用API完成目标？
  - 充分利用已有知识
  - 需要的新知识，越少越好
- 参考RESTful API，一组标准操作方法
  - 不管什么资源，标准操作方法都是相同的
  - REST标准方法也经常实践充分检验

4.1标准方法-概述

标准方法	操作描述	举例
Get	查询已有的资源	GetChatRoom()
List	查询资源集合	ListChatRooms()
Create	创建一个新的资源	CreateMessage()
Update	更新一个已有资源	UpdateUserProfile()
Delete	删除一个已有资源	DeleteChatRoom()
Replace	整体替换一个资源	ReplaceChatRoom()

4.1标准方法-概述

这个模式就结束了么？

- 标准方法，直观明义
- 仍有大量细节隐藏和丢失
  - The devil is often in the details.
  - 删除一个不存在的资源，如何响应？
  - 关注行为，还是强调结果？
  - 针对不存在的资源，执行无权的操作？

4.1标准方法-设计实现

需要实现哪些方法？

- 严格要求实现所有标准方法吗？
  - 保持一致性？
  - 现实场景，并不所有资源都需要实现所有标准方法？
- 一般而言都要实现，除非有明确的合理的不能实现的理由
  - 单例资源，405 Method Not Allowed
  - 资源的某个特定实例，只读文件，403 Forbidden

4.1标准方法-设计实现

幂等性和副作用

- 幂等性idempotence
  - 多次(同一个)操作，结果是一致的
  - 分布式场景，重复请求，应用超时或失败重试，网络抖动
  - 数据丢失，数据损坏，数据不一致
- 副作用side-effect
  - 非预期的影响，没有额外的行为
  - 新建邮件，涉及邮件发送逻辑

4.1标准方法-设计实现

GET方法

- 根据资源标识符获取资源信息
- 是幂等的，且没有任何副作用
  - 不涉及底层数据的变更

4.1标准方法-设计实现

List方法

- 根据查询条件(过滤)返回符合条件的一个资源列表
- 资源列表，可能是资源集合的一个子集，也可能属于某个其他资源
  - eg，ListMessages，查询条件：parent、filter

```
@get("/{parent=chatRooms/*}/messages")
ListMessages(req: ListMessagesRequest): ListMessagesResponse;
```

```
interface ListMessagesRequest {
  parent: string;
  filter: string;
}
```

4.1标准方法-设计实现

List方法

- 访问控制
  - 同样的查询条件，每个人请求的返回结果是否一致？
  - 需要有权限控制，只能返回有权访问的那部分资源
  - 合理组织资源，便于直接判断是否越权，直接返回403 Forbidden错误信息

4.1标准方法-设计实现

List方法

- 结果数量
  - 结果数量，对于页面展示应用非常友好
  - 要平衡实现成本
  - 大规模数据量，分布式存储，统计耗时？
  - 预估统数量，resultCountEstimate
  - 应用场景，feed流

4.1标准方法-设计实现

List方法

- 排序
  - 数据规模，可行性
  - 存储系统，mysql自增ID主键排序
  - 应用场景，必要性
  - 接口排序请求，与系统端排序逻辑

4.1标准方法-设计实现

List方法

- 过滤
  - 鼓励使用过滤，只返回用户关注的结果
  - 极大降低非必要的计算和传输网络带宽
  - 使用通用化的字符串类型表达过滤条件
  - 服务端解析过滤后，再传给存储系统

4.1标准方法-设计实现

Create方法

- 建议使用POST方法
- 标识符
  - 创建时最好由服务端生成标识符
  - 规范标识符的统一生成逻辑
  - 特殊场景由客户端生成的，约定统一的生成逻辑

4.1标准方法-设计实现

Create方法

- 一致性
  - 强一致性，弱一致性
  - 最终一致性，不一致是暂时的，最终要一致
  - 分布式系统，无限水平扩展的一种副作用
  - 从接口设计的可预期角度来看，如果不能保证没有副作用（写后读操作，不能立即查到），最好使用自定义方法来实现

4.1标准方法-设计实现

Update方法

- 建议使用PATCH方法，部分更新
- 更新资源的信息 vs 变更资源的状态
  - 变更状态一般涉及副作用（额外的操作），建议使用自定义方法
  - eg，资源归档操作，除了更新资源的状态，还需要数据转移的操作

4.1标准方法-设计实现

Update方法

- 建议使用PATCH方法，部分更新
- 更新资源的信息
- 特殊的更新：变更资源的状态
  - 变更状态一般涉及副作用（额外的操作），建议使用自定义方法
  - eg，资源归档操作，除了更新资源的状态，还需要数据转移的操作

4.1标准方法-设计实现

Delete方法

- 建议使用DELETE方法
- 幂等性
  - 多次执行，删除同一个资源，如何返回？
  - 删除的行为 还是 结果？
  - 声明式declarative API，命令式imperative API

4.1标准方法-设计实现

Replace方法

- 建议使用PUT方法
- 强Schema场景，整体替换，没有中间状态
- 能否用于创建新资源？
- 能否指定标识符？

4.1标准方法-Trade-offs

- 放弃一些灵活性
- 完全自定义的RPC风格API
- 规范，一致，成本
- 不绝对，二八原则，标准方法解决80%的问题

4.2自定义方法-背景

- 标准方法不是万能的
  - 发送邮件
  - 即时文本翻译
- 是否把多个额外的操作揉合到一个标准方法中？
- 是否要打破标准方法的框架或固有认知？
- 一个方法揉合所有操作，每个操作对应一个方法？

4.2自定义方法-概述

- 不满足标准方法上的严格约束
- 没有约束，完全自由发挥？
- 杂乱发展，将难以维护
- 在整个API设计中，仍需保持某种一致性的约束
  - 标准方法的规范，是所有API都要遵循的
  - 自定义方法的规范，由设计者来制定并遵循

4.2自定义方法-设计实现

方法命名

- 与标准方法相同的命名习惯
- 构成格式：动词 + 名词
- 其他约束也要遵循：介词，复数等

4.2自定义方法-设计实现

副作用

- 是与标准方法最大的差异
- 标准方法的强约束：没有副作用，不涉及额外的操作
- 比如：发送邮件，触发底层操作，更新多个资源

4.2自定义方法-设计实现

定制方法是放在哪里？

- 访问资源上，还是集合上？
- POST /users/1:exportEmails
- POST /user/1/emails:export
- POST /users/1:export
- POST /users/-/emails:archive

4.2自定义方法-设计实现

Stateless场景

- 即时处理请求后返回结果，不涉及其他数据的获取或存储
- FaaS函数即服务，业务逻辑
- 完全无状态，上下文？
  - @post("/text:translate")
  - @post("/{parent=projects/*}/text:translate")
  - @post("/{id=translationModels/*}/text:translate")
  - @post("/translationModels")

4.2自定义方法-Trade-offs

- 与REST设计原则会有冲突
- 在简洁的资源体系下，扩展非标准交互的能力
- 经常被误用或过度使用
  - 不管标准方法是否支持，所有场景都采用自定义方法
  - 难以维护，改善升级

4.3部分更新与检索-背景

- 部分检索
  - 边缘设备处理能力
  - 网络带宽
- 部分更新
  - 数据一致性
  - 数据丢失
  - 细粒度的使用场景

4.3部分更新与检索-概述

- 一种简洁的实现方式，使用字段掩码field mask
  - 能够细粒度描述关注的字段
  - 更新哪些哪些，只返回哪些字段？

4.3部分更新与检索-设计实现

如何传输？

- HTTP协议构成
  - 请求串，请求体，Header
  - GET没有请求体
  - PATCH请求体应该只包括资源部分
- 不同HttpServer服务端，对请求串中数组的处理方式存在差异

/users?fieldMask=1&fieldMask=2

/users?fieldMask[0]=1&fieldMask[1]=2

/users?fieldMask[]=1&fieldMask[]=2

/users?fieldMask=1,2

4.3部分更新与检索-设计实现

接口中的Map或嵌套类型

- 请求的参数不是平铺的，多层数据结构
- fieldmask数据格式约定
  - 使用逗点连接不同的部分
  - 使用星号标识某个嵌套信息中的所有字段
  - Map Key是字符类型
  - Map Key中特殊字符使用反引号字符转义
  - Map Key中反引号字符使用两个反引号转义

```
interface ChatRoom {
  id: string;
  title: string;
  description: string;
  loggingConfig: LoggingConfig;
  settings: Object;
}

interface LoggingConfig {
  maxSizeMb: number;
  maxMessageCount: number;
}
```

- title
- loggingConfig.*
- loggingConfig.maxSizeMb

4.3部分更新与检索-设计实现

列表类型

- 可以与Map类型类似
- 这样做，是否合理？
  - index是否就一定是我们需要的那个？
  - 生成列表时是否是有序的？
  - 这个列表是否被多人修改？
  - 只是获取所有人的姓名？
  - 只是获取任意一个?
  - 能否用于更新？

```
interface ChatRoom {
  id: string;
  title: string;
  description: string;
  administrators: User[];
}

interface User {
  name: string;
  email: string;
  // ...
}
```

- administrators[0].email
- loggingConfig[*].name

4.3部分更新与检索-设计实现

默认值

- 回顾，什么是默认值？
- 部分检索
  - 与标准Get方法默认一致，我们需要列出所有字段么？
  - 如果接口升级，增加字段，所有客户端都需要更新？
  - 引入星号，代表所有字段
- 部分更新
  - 也是默认所有字段？
  - 还是基于输入数据，自动推断？

4.3部分更新与检索-设计实现

隐式field mask

- 对于部分更新，默认与标准update方法一致？Replace？
- 标准update方法使用PATCH
  - 补丁？
- 根据请求数据推断field mask，
  - 提供了哪些字段，就更新哪些字段
  - 某个字段值为null，如何处理？

4.3部分更新与检索-设计实现

更新动态数据结构

- 三种情况：有值，null值，key缺失
- 如何删除某个key？
  - 使用标准replace方式
  - 使用PATCH方法，在filed mask指定的字段，在请求体不存在
  - fieldMask=settings.test&fieldMask=title
  - {“title”: “new title”}

4.3部分更新与检索-设计实现

无效字段

- 静态数据结构，检索或更新不存在的字段？
- 防御性编码策略？
  - 抛出异常，返回错误
- 一致按undefined处理？

4.3部分更新与检索-Trade-offs

- 应用场景约束
  - 最小化非必要数据的传输
  - 精细化数据修改
- 为保持一致性，所有接口通用支持
- 其他实现方法
  - GraphQL

4.4批处理-背景

- 部分更新与检索，是针对单个资源的精细操作
- 另一个方向的功能扩展，针对多个资源进行操作
- 如何像数据库的事务一样，保持原子性？

4.4批处理-概述

- 是否需要完全成熟的通用事务设计模型？
  - 支持必要原子操作
- 在标准方法前期加上Batch
  - BatchDeleteMessages
  - 方法类似，但实现会有差异

4.4批处理-概述

- HTTP方法使用哪种？
  - 类似自定义方法，使用POST？
  - 或类似标准方法
- 如何保持原子性？
  - 批量查询，如果一个字段被删除了，是否整个都失败
  - 能否在多个父资源的进行操作？

4.4批处理-设计实现

原子性

- 原子性的定义？
  - 表示一组操作是原子性的，他们之间不可分割
  - 全部成功或全部失败，没有中间状态
- BatchGetMessages
  - 如果一个资源失败了，整个请求必须是失败的
  - 所有变更都被接受，或一个变更都没被接受
  - 这虽然对批量获取不一定完全适用

4.4批处理-设计实现

结果顺序

- 结果的顺序，与请求的顺序一致
- 结果和请求的对应关系，逻辑判断的复杂性和准确性

4.4批处理-设计实现

常用字段

- 两种策略
  - 请求对象数组，通用性更好，比如针对不同父对象的操作
  - 主键字段数组，更简洁，比如获取或删除指定一组资源
- 策略共用
  - 格外注意
  - 两种方式可能存在冲突
  - APIServer 拒绝执行

4.4批处理-设计实现

跨父资源操作

- 使用对象数组，比共用更优
- 每个请求对象，单独指定父资源

4.4批处理-设计实现

BatchGet操作

- 除了顶层资源，一般都有一个parent指示字段
- 跨父资源，可使用通配符，比如连接字段
- 跨父资源是否合理？
  - 分布式系统中，不同父资源可能存储在不同位置
  - 只要一个失败，则都会失败。可能需要更宽松的保证

4.4批处理-设计实现

BatchGet操作

- 部分检索
  - 统一的fieldmask
- 批操作一般不实现分页，而是限制返回数量

4.4批处理-设计实现

BatchDelete操作

- 使用主键字段数组，使用POST方法
- 除了顶层资源，一般都有一个parent指示字段
- 跨父资源，使用通配符，同时要充分考虑合理性和可行性
- 原子性，删除所有给定的所有资源，或整体失败
  - 删除的概念
  - 如果已经被删除，不再存在，删除操作也需要是失败的

4.4批处理-设计实现

BatchCreate操作

- 请求使用对象数组
- 除了顶层资源，一般都有一个parent指示字段
- 预先是没有主键字段的，这个是API Server端创建的
- 请求与结果的顺序一致性

4.4批处理-设计实现

BatchUpdate操作

- 与BatchCreate基本相似
- 差异，部分更新
  - 部分更新字段需要，每个资源可以不同
  - fieldmash针对单个的资源
- 使用POST方法
  - 与标准Update方法的PATCH不同

4.3批处理-Trade-offs

- 首先把原子性放在首位，即使实现不方便
- 一致性
  - 简单性优于一致性
  - 请求，有时是主键字段，有些是对象

4.5分页-背景

- 资源的大小和数量，量级非常大
  - 通过单次请求和响应获取数据，可能会非常慢或不可行
- 数据切割成可管理的分区，消费者可轻易获取单个分区

4.5分页-概述

- 每次返回一页中的记录，同时返回下一次消费的指针位置
- 引入光标的思想
- 我需要一页 & 我需要页xx

4.5分页-设计实现

引入三个字段

- pageToken
- maxPageSize
- nextPageToken

4.5分页-设计实现

Page size

- 最大值还是准确值？
- 默认值
  - 视资源大小而定，一般10条
  - 配置指定默认值
  - 一致性
- 上界和下界
  - 负数，和超过一定数值，可以直接拒绝

4.5分页-设计实现

Page tokens

- a cursor
- 如何表示终止
  - 返回一页数据不满？
  - 返回空页？
  - 考虑耗时因素，可能为空页，但有nextPageToken

4.5分页-设计实现

Page tokens

- 一致性
  - 新增和删除数据，下一页返回结果与当前页是相同的
  - 依赖使用的存储系统
  - 避免使用绝对的偏移量，使用上次的结果作为相对偏移量

4.5分页-设计实现

Total count

- 是个好主意，而且会带来友好的用户体验
- 总数非常大？几乎不可能返回准确值
- 耗时更重要
  - 不提供总数，或者返回预估值，或者有明确的需求

4.5分页-设计实现

单一大资源，如何分页？

- 类似机制
- pageToken，maxBytes

4.4分页-Trade-offs

- 双向分页
  - 当前模式难以支持
  - 一般情况下，不是必须的功能
  - 一种可能方案，构建缓存
- 任意指定窗口
  - 直接访问特定分页，在人机接口常用
  - 可以通过过滤或排序，

4.6长耗时操作-背景

- 复杂操作，处理大数据量
- 用于即时操作的设计方法不一定适用
- 能够支持异步调用的API，非常有价值

4.6长耗时操作-概述

- 类似Future或Promise机制
  - 启动任务，不阻塞，立即返回一个Pormise对象
  - 在Promis对象上等待，或注册回调函数
- 有一些差异
  - 一种资源，需要持久化
  - 还有一些元数据，进度，开始或完成时间，当前执行的动作
  - 需要一种发现和管理LROs的方法

4.6长耗时操作-设计实现

两个部分

- 定义Operation资源
  - 结果类型 和 元数据
- 发现和管理LROs的API接口

4.6长耗时操作-设计实现

LRO资源

- 作为一种资源，能够交互
- 最终返回结果
  - 明确的表示结果
  - 除了正常结束的结果后，还需要考虑有错误发生或者未成功完成
  - 还有一种情况，只是标识成功完成，没有其他结果信息
  - 所以结果字段是可选的，是正常的结果类型（泛型），或者操作错误类型

4.6长耗时操作-设计实现

LRO资源

- 状态标识
  - done标识字段，是否已结束
  - pending，resolved，rejected
- 元数据信息
  - 在结束前，提供一些任务的动态信息
  - 进度信息，预估剩余时间
- 关注点：最终返回结果，和元数据信息，都可能是非必须的

4.6长耗时操作-设计实现

LRO 资源的层级

- 集中式顶层资源集合

4.6长耗时操作-设计实现

LRO解析

- 如果基于一个LRO资源，获取最终返回结果
- Polling，轮训，GetOperation
  - 客户端，检查频次，停止检查
  - 不能立即知道结果
- Wating，等待，WaitOperation
  - 一直维持链接，一旦结束，立即知道结果
  - 实现方式，服务端的轮训
  - 如果链接丢失，可以再使用轮训方式

4.6长耗时操作-设计实现

错误处理

- 在HTTP请求中，错误处理使用http 错误码
- LRO不能使用
  - 返回的http错误码，不能区分是操作的结果，还是GetOperation的处理结果
  - http错误码仅用来表示资源操作的处理结果
- LRO的处理结果，放在最终返回结果字段OperationError
  - 错误码&错误信息，给计算机还是人使用
  - 除了错误类型，额外信息，最好是结构化的，机器可识别的方式

4.6长耗时操作-设计实现

进度监控

- 使用MetaDataT来表达
  - 任务开始时间，任务进度
  - 预估剩余时间，已处理字节数量
- 应用
  - 每次轮训时，获取进度

4.6长耗时操作-设计实现

取消操作

- 为什么取消？
  - 创建时，使用错误数据
- 自定义方法，CancelOperation
  - 需要等待完成，阻塞
  - 中间结果要清理，回到初始状态
  - 有些无法清理的，需要反馈出来，通过元数据信息
  - 并不是所有操作都可取消

4.6长耗时操作-设计实现

暂停和恢复操作

- 并不是所有操作都可暂停或恢复？
  - 只有合理和可行的场景
- 自定义方法，PauseOperation，ResumeOperation
  - 元数据中新增一个布尔字段，paused
  - 同样需要等待完成，阻塞

4.6长耗时操作-设计实现

管理操作

- 创建和执行是分离的
  - 客户端崩溃了，操作指针和状态就丢失了，无法重启后恢复轮训
  - 虽然可以通过操作标识符获取信息，但无法发现正在执行的操作，或者探查历史的操作
- 通过ListOperations
  - 支持过滤，比如查询暂停的操作列表d
  - 没有标识符的情况

4.6长耗时操作-设计实现

持久化

- LROs资源的区别
  - 不是直接创建，隐性创建
  - 是其他行为的副产品
  - 起初很重要，一旦完成后变的不重要

4.6长耗时操作-设计实现

持久化

- 持久化策略
  - 与其他常规资源保持一致
  - 滚动窗口，比如30天有效期，有效期外，会被删除
  - 其他策略，操作对应的资源删除了，二级有效期，归档，删除归档

4.6长耗时操作-Trade-offs

- 最简单的办法就是等待
  - 可能不能很好适应分布式系统
  - 一个微服务初始化任务，另一个监控进度
  - 监控进度，意味着一直阻塞等待
  - 如果链接短了，就可能会丧失恢复的能力
- 复杂，概念多，引起混淆（Rerunnable jobs）

4.7重复运行作业-背景

- 虽然有异步调用的API，但仍需客户端触发执行
  - 这针对按需执行的场景，非常适用
- 还有三类场景，不能很好的支持
  - 异步执行，但每次调用需要提供所有相关配置
  - 按需执行模型，混合两种权限：执行方法的能力，和配置参数的能力
  - 按某种循环计划的方式自动执行

4.7重复运行作业-概述

- 引入作业job概念
- job是一种特殊的资源类型
  - 配置过程
  - 执行过程
- 如何解决三个问题的？
  - 一次配置，多次执行
  - 划分两个方法，更便于控制权限
  - 在调度系统中，只调用执行接口即可，不需要关注大量繁琐的配置

4.7重复运行作业-设计实现

Job作业资源

- 标准方法
- 并非所有方法都要支持
  - 比如Job是不可变更的，可忽略update方法

4.7重复运行作业-设计实现

自定义方法run

- 入参只有一个，不需要配置信息

4.7重复运行作业-设计实现

Execution作业执行资源

- 为什么需要？
  - 与LRO资源持久化策略不一致
  - 根据不同的job资源，记录不同的信息，这与LRO不同
- 标准方法
  - Get，List，Delete
- 资源层级
  - 属于单一的Job类型下

4.7重复运行作业-设计实现

Execution作业执行资源

- 为什么需要？
  - 与LRO资源持久化策略不一致
  - 根据不同的job资源，记录不同的信息，这与LRO不同
- 标准方法
  - Get，List，Delete
- 资源层级
  - 属于单一的Job类型下

4.7重复运行作业-Trade-offs

- 问题集，多种解决方案
- 比如针对权限问题，可以设计高级的权限系统
  - API方法 + 方法参数都进行检查控制
  - 缺点，实现逻辑复杂
- 比如执行资源，可以只保持Operation资源，
  - 作为输出的引用
  - 缺点，过滤Operation资源获取Job关注的信息

4.8导入导出-背景

- 虽然已经有资源输入和输出的办法，单个资源或批的方式
- 但有时我们有一些序列化的数据，需要导入
  - 一种方法是写导入程序，解析数据成资源，然后调用标准方法写入API
  - 不方便，链路长
  - 存储系统与API Server服务器相邻，而导入程序在另一个机房
- 对于数据导出，也有类似问题
- 这类场景还是常见的

4.8导入导出-概述

- 使用两个自定义方法：import，export
- 模块构成
  - 数据获取
  - 数据处理
- 流程
  - 启动import
  - 获取数据
  - 转换数据
  - 更新资源

4.8导入导出-概述

- 挑战
  - 各类存储系统
  - 各类序列化格式
  - 多种数据处理，压缩，加密等
  - 如何灵活配置支持新的系统和格式

4.8导入导出-概述

- 其他问题
  - 如何处理失败
  - 是否需要重试
  - 数据的一致性
  - 导入时如何处理数据融合

4.8导入导出-设计实现

自定义方法

- import&export
  - 返回类型是LRO

4.8导入导出-设计实现

存储系统交互

- 与各种存储系统交互的配置信息如何组织？
  - 一种方式是单个schema，保存所有配置
  - 使用接口，针对不同存储系统特定实现
- 针对import和export，为什么需要不同的接口？
  - 首先是有相似性的，交互的两个主体相同
  - 但也有差异
  - 允许两个接口独立变更，

4.8导入导出-设计实现

资源与二进制的转换

- 已经有序列化API资源的机制，比如JSON格式
  - import&export场景不一定完全适用
  - 比如每行一个json，而不是一个整体的json array
  - 也能是完全不同的格式，比如csv等
- 除了格式不同，可能还需要额外的数据处理
  - 压缩或加密
  - 文件切分

4.8导入导出-设计实现

资源与二进制的转换

- 配置InputConfig、OutputConfig
- 关注点
  - 不是大量配置信息本身，
  - 而是将存储系统的连接信息，与数据处理的方式，分开配置
  - 又是一种松耦合的设计

4.8导入导出-设计实现

一致性

- 在导出数据时，首先要保障导出所有或满足条件的所有的数据
  - 这在数据量很少的场景下，很容易支持
  - 但在大数据量，或耗时很长的导出过程，就很困难

4.8导入导出-设计实现

一致性

- 如果在导出的过程中，资源可能发生变更了，怎么办？
  - 一种是依赖底层存储系统的快照和事务能力
  - 读取特定时间点的所有数据
  - 但并不是所有存储系统都支持快照功能，数量量越大，越不容易支持
  - 另一种办法，接受现实，不可能是特定时间点的所有数据，但尽最大可能是精确的数据，
  - 导出数据，可能是不存在的一种数据状态
  - 如果不能接受的话，需要在导出期间，禁止变更

4.8导入导出-设计实现

一致性

- 导出数据，与备份数据，差异
  - 备份是一种快照
  - 导出是通过数据检索的方式，转移到外部存储系统

4.8导入导出-设计实现

主键与冲突

- 挑战
  - 导入数据主键已经存在
  - 导出后再次导入，但不同的父资源
  - 有些API不允许用户指定标识符

4.8导入导出-设计实现

主键与冲突

- 回归到导入&导出的初衷
  - 只是作为API和外部存储系统的中间桥梁，替代了编写应用程序实现数据转移
  - 这意味着，需要与编写应用程序的方式保持一致
  - 根据提供的数据，通过batchcreate方法创建一批对象
  - 如果不允许用户指定的标识符，那在import数据时，需要忽略标识符字段
  - 导出后再次导入，但不同的父资源
  - 有些API不允许用户指定标识符
  - 同时，如果后续需要判断数据来源，那就在导出数据时，保留标识符资源
  - 但这并不是意味着，多次导入同一份数据会创建大量重复资源

4.8导入导出-设计实现

主键与冲突

- 导入&导出不能直接用于备份和恢复
  - 备份和恢复，需要更严苛的数据检查
  - 备份时需要一致性的快照
  - 导入时是完整的替换

4.8导入导出-设计实现

处理关联资源

- 针对父资源的导入&导出，是否包含所有的子资源？
- 仍然回归初衷
  - 只是list操作或batch操作的一种等价物
  - 就应像batch操作那种，只针对一种资源
  - 不应因为合理，就额外增加功能
- 同时跟备份和恢复，需要再次划清界限

4.8导入导出-设计实现

失败与重试

- 易于发生错误，处理失败充满挑战
  - 如何处理失败，能否重试？
- 两种问题原因
  - API服务的问题
  - 存储系统的问题

4.8导入导出-设计实现

失败与重试

- 导出失败
  - 一般而言，每次导出是独立的，所以重试是安全的
  - 但也有几个问题需要重视
  - 每次导出执行可能不是相同的
  - 上一次导出失败的是否删除还是保留稍后手动处置？
  - 如果是存储系统的问题，比如磁盘空间不足

4.8导入导出-设计实现

失败与重试

- 导入失败
  - 不能像导出那种，无法简单的直接重试
  - 如果是因为数据校验，而不是网络问题，再次重试都会发生同样的问题
  - 即使是因为偶发性问题，也不能直接重试，因为前一次导入可能创建了一部分新资源，导致数据重复

4.8导入导出-设计实现

失败与重试

- 导入失败处置策略
  - 简单的办法是在一个事务中执行导入操作
  - 但并不是所有存储系统都支持
  - 而且在同一个事务导入大批量数据，也不可行
  - 另一办法是支持请求去重
  - 不一定是资源标识符，可以是每条记录的导入请求id
  - 导出时，可配置是否序列化到输出数据中

4.8导入导出-Trade-offs

- 这个模式定位非常窄和具体
  - 只是作为API和外部存储系统的中间桥梁，替代了编写应用程序实现数据转移
- 这导致有两个缺点
  - 只能用于转移一种资源类型，否则就不是简单的纯粹的一种数据转移，而是包含业务逻辑
  - 容易与数据备份和恢复功能混淆，没有一致性保障

QA