分类依据

总述：

好的API接口对于用户来说必须满足以下几个点：

易学习：有完善的文档及提供尽可能多的示例和可copy－paste的代码，像其他设计工作一样，你应该应用最小惊讶原则。

易使用：没有复杂的程序、复杂的细节，易于学习；灵活的API允许按字段排序、可自定义分页、 排序和筛选等。一个完整的API意味着被期望的功能都包含在内。

难误用：对详细的错误提示，有些经验的用户可以直接使用API而不需要阅读文档。

而对于开发人员来说，要求又是不一样的：

易阅读：代码的编写只需要一次一次，但是当调试或者修改的时候都需要对代码进行阅读。

易开发：个最小化的接口是使用尽可能少的类以及尽可能少的类成员。这样使得理解、记忆、调试以及改变API更容易

## 权限控制

介绍

严格来说，服务授权包含鉴权（Authentication）和授权（Authorization）两部分。鉴权解决的是调用方身份识别的问题 。授权解决的是调用是否被允许的问题 。两者一先一后，缺一不可。为避免歧义，如不特殊指明，下文所述授权都是宽泛意义上的授权，即包含了鉴权。

常见的服务授权有三种，简单授权，协议授权和中央授权。

1.简单授权：服务提供方并不进行真正的授权，而是依赖于外部环境进行自动授权，比如IP地址白名单，内网域名等。

2.协议授权：服务提供方和服务调用方事先约定一个密钥，服务调用方每次发起服务调用请求时，用约定的密钥对请求内容进行加密生成鉴权头（包含调用方唯一识别ID），服务提供方收到请求后，根据鉴权头找到相应的密钥对请求进行鉴权，鉴权通过后再决定是否授权此次调用。

3.中央授权：引入独立的授权中心，服务调用方每次发起服务调用请求时，先从授权中心获取一个授权码，然后附在原始请求上一起发给服务提供方，提供方收到请求后，先通过授权中心将授权码还原成调用方身份信息和相应的权限列表，然后决定是否授权此次调用。

一般来说，简单授权在业务规则简单、安全性要求不高的场景下用的比较多。而协议授权，比较适用于点对点或者C/S架构的服务调用场景，比如Amazon S3 API。对于网状结构的微服务而言，中央授权是三种方式中最适合也是最灵活的选择：

* 简化了服务提供方的实现，让提供方专注于权限设计而非实现。
* 提供了一套独立于服务提供方和服务调用方的授权机制，无需重新发布服务，只要在授权中心修改服务授权规则，就可以影响后续的服务调用。

方案选择

方案选择包括两方面需求：其一是认证与鉴权，对于请求的用户身份的授权以及合法性鉴权；其二是API级别的操作权限控制，这个在第一点之后，当鉴定完用户身份合法之后，对于该用户的某个具体请求是否具有该操作执行权限进行校验。

**需求一认证鉴权:**

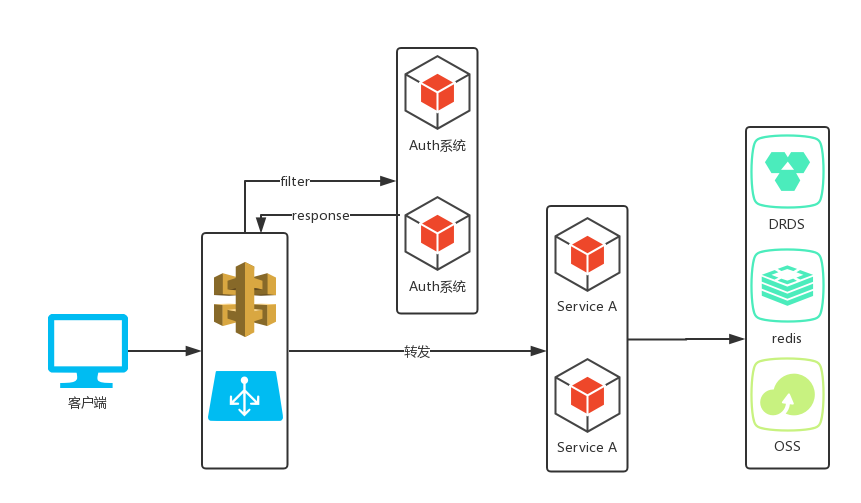
方案1：分布式Session方案

分布式会话方案原理主要是将关于用户认证的信息存储在共享存储中，且通常由用户会话作为 key 来实现的简单分布式哈希映射。当用户访问微服务时，用户数据可以从共享存储中获取。在某些场景下，这种方案很不错，用户登录状态是不透明的。同时也是一个高可用且可扩展的解决方案。这种方案的缺点在于共享存储需要一定保护机制，因此需要通过安全链接来访问，这时解决方案的实现就通常具有相当高的复杂性了。

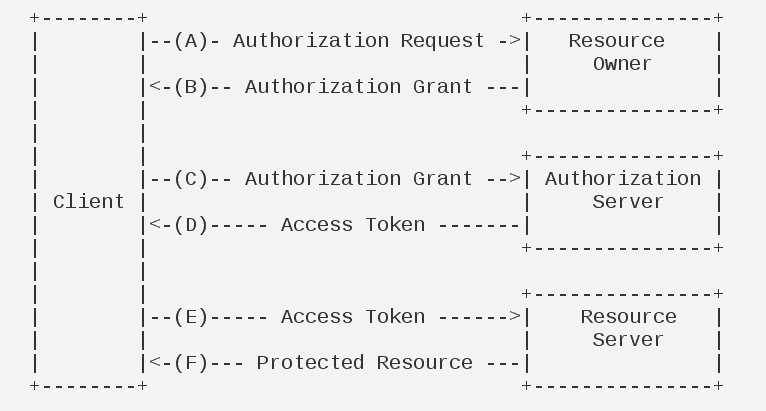
方案2：基于OAuth2 Token方案

随着 Restful API、微服务的兴起，基于Token的认证现在已经越来越普遍。Token和Session ID 不同，并非只是一个 key。Token 一般会包含用户的相关信息，通过验证 Token 就可以完成身份校验。用户输入登录信息，发送到身份认证服务进行认证。AuthorizationServer验证登录信息是否正确，返回用户基础信息、权限范围、有效时间等信息，客户端存储接口。用户将 Token 放在 HTTP 请求头中，发起相关 API 调用。被调用的微服务，验证Token。ResourceServer返回相关资源和数据。

选择：方案2



分布式架构中鉴权



Oauth2运行流程

优势

服务端无状态：Token 机制在服务端不需要存储 session 信息，因为 Token 自身包含了所有用户的相关信息。

性能较好，因为在验证 Token 时不用再去访问数据库或者远程服务进行权限校验，自然可以提升不少性能。

现在很多应用都是同时面向移动端和web端，OAuth2 Token机制可以支持移动设备。

OAuth2与Spring Security结合使用，有提供很多开箱即用的功能，大多特性都可以通过配置灵活的变更。

最后一点，也很重要，Spring Security OAuth2的文档写得较为详细。

需求二操作权限控制:

方案1：shiro

Shiro是一个强大而灵活的开源安全框架，能够非常清晰的处理认证、授权、管理会话以及密码加密。Shiro很容易入手，上手快控制粒度可糙可细。自由度高，Shiro既能配合Spring使用也可以单独使用。

方案2：Spring Security

Spring社区生态很强大。除了不能脱离Spring，Spring Security具有Shiro所有的功能。而且Spring Security对Oauth、OpenID也有支持,Shiro则需要自己手动实现。Spring Security的权限细粒度更高。但是Spring Security太过复杂。

Oauth2介绍，待补充

其他：

添加白名单：校验IP是否在白名单

使用SSL（https）来提供URL：HTTPS在HTTP的基础上添加了SSL安全协议，自动对数据进行了压缩加密，在一定程序可以防止监听、防止劫持、防止重发，安全性可以提高很多。

采用POST作为传输方式

对传输参数进行非对称加密

## 版本控制

方式一：URI 版本控制

每次修改 Web API 或更改资源的架构时，向每个资源的 URI 添加版本号。 以前存在的 URI 应像以前一样继续运行，并返回符合原始架构的资源。（如 <http://adventure-works.com/v2/customers/3>:）

好处：

直接可以在URI中直观的看到API版本，

可以直接在浏览器的查看各个版本API的结果

坏处：

版本号在URI中破坏了REST的HATEOAS（hypermedia as the engine of application state）规则。[1]版本号和资源之间并无直接关系。

方式二：添加头信息

在API请求header中添加Accept字段。 Accept的作用是客户端指出响应可以接受的媒体类型 。（如Accept:application/json; version=v2）

好处：

遵循了REST的设计风格。

坏处：

版本不直观，需要能设置header的client才能调用查看该API的效果。

方案考虑：将接口版本分为主版本与次版本。主版本体现在URI中，此版本小改动体现在头信息中。[2]

## 流控

介绍

由于业务应用系统的负载能力有限，为了防止非预期的请求对系统压力过大而拖垮业务应用系统时，需要对大流量进行控制，即为流控。

服务接口的流量控制策略：分流、降级、限流等。此处介绍限流策略，虽然降低了服务接口的访问频率和并发量，却换取服务接口和业务应用系统的高可用。

业界流控算法：

漏桶(Leaky Bucket)算法



水(请求)先进入到漏桶里，漏桶以一定的速度出水(接口有响应速率),

当水流入速度过大会直接溢出(访问频率超过接口响应速率), 然后就拒绝请求,

可以看出漏桶算法能强行限制数据的传输速率.

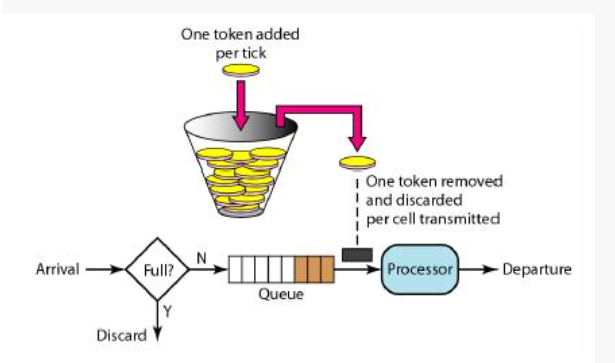
有两个变量：

* 一个是桶的大小,支持流量突发增多时可以存多少的水(burst),
* 另一个是水桶漏洞的大小(rate)。

漏桶算法的问题：

* 水满了直接不让进、丢弃
* 桶容量问题
* 漏出速度控制问题

令牌桶算法(Token Bucket)



随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)

往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),

如果桶已经满了就不再加了. 新请求来临时,

会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.

令牌添加速度支持动态变化，实时控制处理的速率.

令牌桶算法的问题：

* 令牌满了，请求无法处理
* 桶容量问题
* 令牌添加速度控制问题

计数器

主要用来限制总并发数，比如数据库连接池、线程池、并发数；只要全局总请求数超过设定的阀值则进行限流，是简单粗暴的总数量限流，而不是平均速率限流。

**漏桶和令牌桶算法对比**

* 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；
* 漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；
* 令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；
* 漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；
* 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；
* 两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。[3]

实现

1.Redis+Lua实现

简陋的设计思路：假设一个用户（用IP判断）每分钟访问某一个服务接口的次数不能超过10次，那么我们可以在Redis中创建一个键，并此时我们就设置键的过期时间为60秒，每一个用户对此服务接口的访问就把键值加1，在60秒内当键值增加到10的时候，就禁止访问服务接口。在某种场景中添加访问时间间隔还是很有必要的。

1）使用Redis的incr命令，将计数器作为Lua脚本

1. local current
2. current = redis.call("incr",KEYS[1])
3. if tonumber(current) == 1 then
4. redis.call("expire",KEYS[1],1)
5. end

Lua脚本在Redis中运行，保证了incr和expire两个操作的原子性。

2）使用Reids的列表结构代替incr命令

1. FUNCTION LIMIT\_API\_CALL(ip)
2. current = LLEN(ip)
3. IF current > 10 THEN
4. ERROR "too many requests per second"
5. ELSE
6. IF EXISTS(ip) == FALSE
7. MULTI
8. RPUSH(ip,ip)
9. EXPIRE(ip,1)
10. EXEC
11. ELSE
12. RPUSHX(ip,ip)
13. END
14. PERFORM\_API\_CALL()
15. END

Rate Limit使用Redis的列表作为容器，LLEN用于对访问次数的检查，一个事物中包含了RPUSH和EXPIRE两个命令，用于在第一次执行计数是创建列表并设置过期时间，RPUSHX在后续的计数操作中进行增加操作。[5]

2. guava

使用guava提供工具库里的RateLimiter类(内部采用令牌捅算法实现)进行限流 。

[6]

3.Hystrix

Hystrix是Netflix开源的一款容错系统，能帮助使用者写出具备强大的容错能力和鲁棒性的程序。[7]

在分布式环境中，不可避免地有许多服务依赖将失败，尤其现在流行的微服务。 Hystrix是一个库，可以通过线程隔离、熔断、服务降级等措施来帮助您控制这些分布式服务之间的交互。[9]

Hystrix提供如下机制来提升分布式系统的可靠性：

保护通过第三方客户端API依赖访问，控制其延迟和故障

阻止级联故障和"雪崩效应"

提供熔断机制，快速失败和恢复

失败回调和优雅降级机制

近实时检测、报警和KPI指标展示

当我们使用了Hystrix时，Hystrix将所有的外部调用都封装成一个HystrixCommand或者HystrixObservableCommand对象，这些外部调用将会在一个独立的线程中运行。我们可以将出现问题的服务通过熔断、降级等手段隔离开来，这样不影响整个系统的主业务。

实现

[8]

## 幂等性

介绍：

WEB 资源或 API 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。幂等性是系统的接口对外一种承诺(而不是实现), 承诺只要调用接口成功, 外部多次调用对系统的影响是一致的。幂等性是分布式系统设计中的一个重要概念，对超时处理、系统恢复等具有重要意义。声明为幂等的接口会认为外部调用失败是常态, 并且失败之后必然会有重试。例如，在因网络中断等原因导致请求方未能收到请求返回值的情况下，如果该资源具备幂等性，请求方只需要重新请求即可，而无需担心重复调用会产生错误。实际上，我们常用的 HTTP 协议的方法是具有幂等性语义要求的，比如：get 方法用于获取资源，不应有副作用，因此是幂等的；post 方法用于创建资源，每次请求都会产生新的资源，因此不具备幂等性；put 方法用于更新资源，是幂等的；delete 方法用于删除资源，也是幂等的。

实现方案：

0.全局唯一ID

如果使用全局唯一ID，就是根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。如果不存在则把全局ID，存储到存储系统中，比如数据库、redis等。如果存在则表示该方法已经执行。

从工程的角度来说，使用全局ID做幂等可以作为一个业务的基础的微服务存在，在很多的微服务中都会用到这样的服务，在每个微服务中都完成这样的功能，会存在工作量重复。另外打造一个高可靠的幂等服务还需要考虑很多问题，比如一台机器虽然把全局ID先写入了存储，但是在写入之后挂了，这就需要引入全局ID的超时机制。

使用全局唯一ID是一个通用方案，可以支持插入、更新、删除业务操作。但是这个方案看起来很美但是实现起来比较麻烦，下面的方案适用于特定的场景，但是实现起来比较简单。

1.MVCC 方案

　　多版本并发控制，乐观锁的一种实现。该策略主要使用 update with condition（更新带条件来防止）来保证多次外部请求调用对系统的影响是一致的。在系统设计的过程中，合理的使用乐观锁，通过 version 或者 updateTime（timestamp）等其他条件，来做乐观锁的判断条件，这样保证更新操作即使在并发的情况下，也不会有太大的问题.

例如

update tableName set name=#name#,version=version+1 where version=#version#

　　在更新的过程中利用 version 来防止，其他操作对对象的并发更新，导致更新丢失。为了避免失败，通常需要一定的重试机制。

2.去重表

在插入数据的时候，插入去重表，利用数据库的唯一索引特性，保证唯一的逻辑。

比如在支付场景中，如果一个订单只会支付一次，所以订单ID可以作为唯一标识。这时，我们就可以建一张去重表，并且把唯一标识作为唯一索引，在我们实现时，把创建支付单据和写入去去重表，放在一个事务中，如果重复创建，数据库会抛出唯一约束异常，操作就会回滚。

3.悲观锁

select for update，整个执行过程中锁定该订单对应的记录。注意：这种在 DB 读大于写的情况下尽量少用。

4. select + insert

　　并发不高的后台系统，或者一些任务 JOB，为了支持幂等，支持重复执行，简单的处理方法是，先查询下一些关键数据，判断是否已经执行过，在进行业务处理，就可以了。注意：核心高并发流程不要用这种方法。

5.状态机幂等

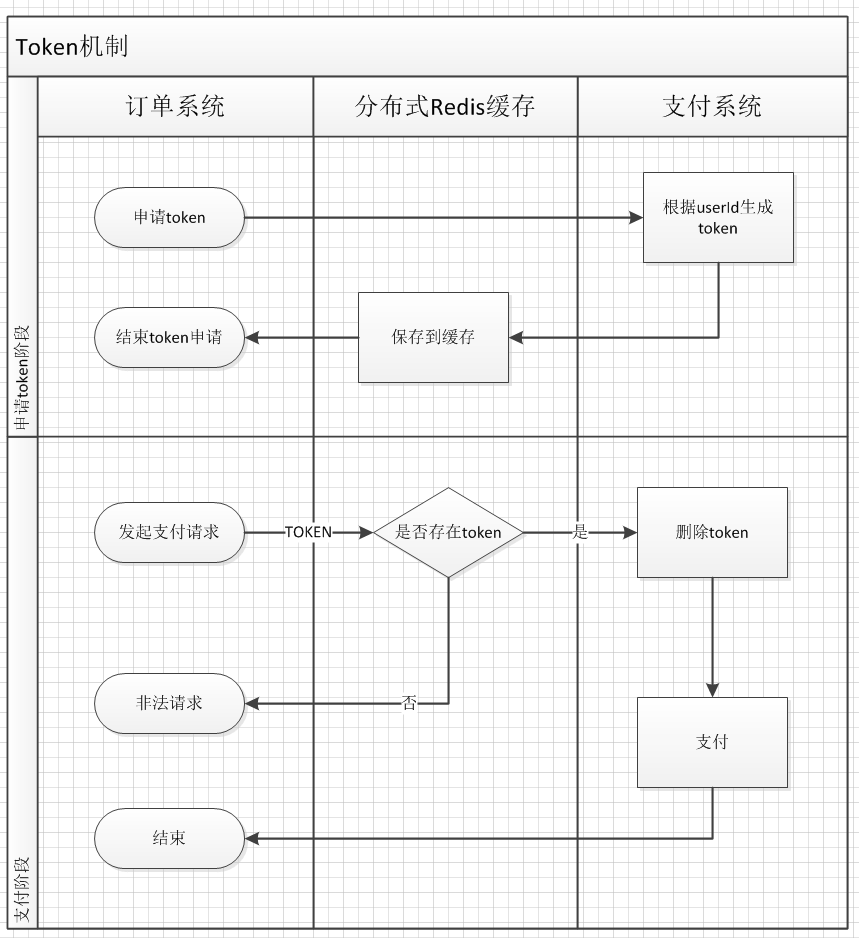
在设计单据相关的业务，或者是任务相关的业务，肯定会涉及到状态机，就是业务单据上面有个状态，状态在不同的情况下会发生变更，一般情况下存在有限状态机，这时候，如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

比如就会订单的创建和付款，订单的付款肯定是在之前，这时我们可以通过在设计状态字段时，使用int类型，并且通过值类型的大小来做幂等，比如订单的创建为0，付款成功为100。付款失败为99

在做状态机更新时，我们就这可以这样控制

update `order` set status=#{status} where id=#{id} and status<#{status}

1. token 机制，防止重复提交[10]



　　这种方式分成两个阶段：申请token阶段和支付阶段。  
第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。  
第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。  
实际上这里的token是一个信物，支付系统根据token确认，你是你妈的孩子。不足是需要系统间交互两次，

解决办法：

集群环境：采用 token 加 redis（redis 单线程的，处理需要排队）

单 JVM 环境：采用 token 加 redis 或 token 加 jvm 内存

处理流程：

数据提交前要向服务的申请 token，token 放到 redis 或 jvm 内存，token 有效时间

提交后后台校验 token，同时删除 token，生成新的 token 返回

token 特点:要申请，一次有效性，可以限流

对外提供接口的 api 如何保证幂等

　　如银联提供的付款接口：需要接入商户提交付款请求时附带：source 来源，seq 序列号。source+seq 在数据库里面做唯一索引，防止多次付款，(并发时，只能处理一个请求)

## 响应

整体数据格式：

Json

<http://www.cnblogs.com/wenzhengs/p/8052001.html>

结构统一封装:

如：

{

“header”: {

“status”: “success”,

“errorCode”:0

},

“response”: {

实际数据

}

}

单个数据格式：

使用多数API中使用的表示相同含义的单词

通过尽可能少的单词来表示

使用多个单词时，整个API连接单词的方法要统一(驼峰法，蛇形法，脊柱法)

尽可能不用奇怪的缩略语

注意单复数形式

ID等巨大数值通常用字符串形式

出错信息：

通过状态码表示出错信息

向客户端返回详细的出错信息(填入响应消息首部/放入消息体)

//放入消息体

{

"status": 404,

"code": 40483,

"message": "Oops! It looks like that file does not exist.",

"developerMessage": "File resource for path /uploads/foobar.txt does not exist. Please wait 10 minutes until the upload batch completes before checking again.",

"moreInfo": "http://www.mycompany.com/errors/40483",

"requestId": "x3kdsa32k23ds32e"

}

status

Status的内容与HTTP状态码内容相同，这个字段的存在，使得错误信息自包含，客户端只需要解析HTTP响应的body部分，就可以获取所有跟这次出错相关的信息。

code

自定义错误码。自定义错误码的长度和个数都可以自己定义，这样就突破了HTTP状态码的个数限制。例子中的错误码是40483，其中404代表了请求的资源不存在，而83则制定了这次出错，具体是哪一种资源不存在。

message

用户可理解的错误信息，应当根据用户的locale信息返回对应语言的版本。这个错误信息意在返回给使用客户端的用户阅读，不应该包含任何技术信息。有了这个字段，客户端的开发者在出错时，能够展示恰当的信息给最终用户。

developerMessage

该出错的详细技术信息，提供给客户端的开发者阅读。可以包含Exception的信息、StackTrace，或者其它有用的技术信息。

moreInfo

给出一个URL，客户端开发站访问这个URL可以看到更详细的关于该种出错信息的描述。在该URL展示的网页中，可以包含该出错信息的定义，产生原因，解决办法等等。

requestId

请求ID，服务为每一个请求唯一生成一个请求ID，当客户端开发者无法自助解决问题时，可以联络服务开发者，同时提供该请求ID。一个好的服务，服务开发者应当可以根据此ID，定位到该次请求的所有相关log，进而定位问题，解决问题。

## 文档