# 什么是幂等性

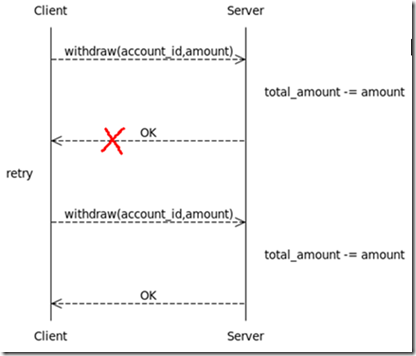
抄写一段数学上的定义：f(f(x))=f(x)，即x被函数f作用一次和作用无限次的结果是一样的。幂等性应用在软件系统中，我把它简单定义为：某个函数或者某个接口使用相同参数调用一次或者无限次，其造成的结果是一样的。在实际应用中，一般针对接口进行幂等性设计。举个例子，在系统中，调用方A调用系统B的接口进行用户扣费操作时，由于网络不稳定，A重试了N次该请求，那么不管B是否接收到多少次请求，都应该保证只会扣除该用户一次费用。

# 分布式事务VS幂等设计

为什么需要幂等性呢？我们先从一个例子说起，假设有一个从账户取钱的远程API（可以是HTTP的，也可以不是），我们暂时用类函数的方式记为：

bool withdraw(account\_id, amount)

withdraw的语义是从account\_id对应的账户中扣除amount数额的钱；如果扣除成功则返回true，账户余额减少amount；如果扣除失败则返回false，账户余额不变。值得注意的是：和本地环境相比，我们不能轻易假设分布式环境的可靠性。一种典型的情况是withdraw请求已经被服务器端正确处理，但服务器端的返回结果由于网络等原因被丢掉了，导致客户端无法得知处理结果。如果是在网页上，一些不恰当的设计可能会使用户认为上一次操作失败了，然后刷新页面，这就导致了withdraw被调用两次，账户也被多扣了一次钱。如图1所示：



这个问题的解决方案之一是采用分布式事务，通过引入支持分布式事务的中间件来保证withdraw功能的事务性。分布式事务的优点是对于调用者很简单，复杂性都交给了中间件来管理。缺点则是一方面架构太重量级，容易被绑在特定的中间件上，不利于异构系统的集成；另一方面分布式事务虽然能保证事务的ACID性质，而却无法提供性能和可用性的保证。

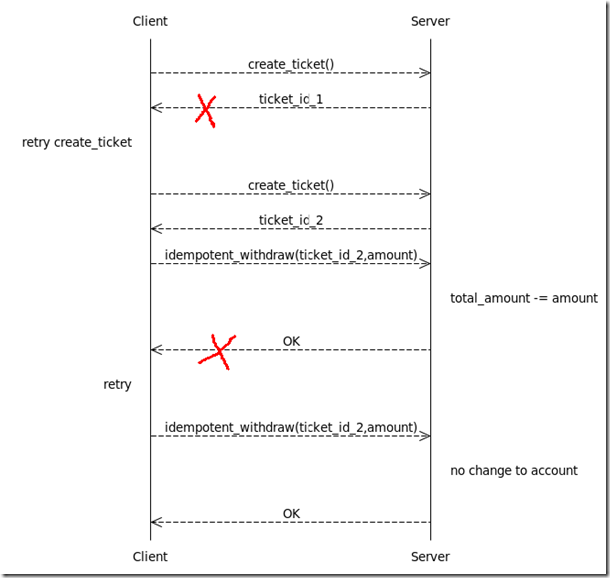
另一种更轻量级的解决方案是幂等设计。我们可以通过一些技巧把withdraw变成幂等的，比如：

int create\_ticket()

bool idempotent\_withdraw(ticket\_id, account\_id, amount)

create\_ticket的寓意是获取一个服务器端生成的唯一处理号ticket\_id，它将用于标识后续的操作。idempotent\_withdraw和withdraw的区别在于关联了一个ticket\_id，一个ticket\_id表示的操作至多只会被处理一次，每次调用都将返回第一次调用时的处理结果。这样，idempotent\_withdraw就符合幂等性了，客户端就可以放心地多次调用。

基于幂等性的解决方案中一个完整的取钱流程被分解成了两个步骤：1.调用create\_ticket()获取ticket\_id；2.调用idempotent\_withdraw(ticket\_id, account\_id, amount)。虽然create\_ticket不是幂等的，但在这种设计下，它对系统状态的影响可以忽略，加上idempotent\_withdraw是幂等的，所以任何一步由于网络等原因失败或超时，客户端都可以重试，直到获得结果。如图2所示：



和分布式事务相比，幂等设计的优势在于它的轻量级，容易适应异构环境，以及性能和可用性方面。在某些性能要求比较高的应用，幂等设计往往是唯一的选择。

# HTTP的幂等性

HTTP协议本身是一种面向资源的应用层协议，但对HTTP协议的使用实际上存在着两种不同的方式：一种是RESTful的，它把HTTP当成应用层协议，比较忠实地遵守了HTTP协议的各种规定；另一种是SOA的，它并没有完全把HTTP当成应用层协议，而是把HTTP协议作为了传输层协议，然后在HTTP之上建立了自己的应用层协议。本文所讨论的HTTP幂等性主要针对RESTful风格的，不过正如上一节所看到的那样，幂等性并不属于特定的协议，它是分布式系统的一种特性；所以，不论是SOA还是RESTful的Web API设计都应该考虑幂等性。下面将介绍HTTP GET、DELETE、PUT、POST四种主要方法的语义和幂等性。

HTTP GET方法用于获取资源，不管调用多少次接口，结果都不会改变，所以是幂等的。

GET /tickets # 获取ticket列表

GET /tickets/12 # 查看某个具体的ticket

HTTP POST方法用于创建资源，每调用一次，都会产生一个新的资源，所以是非幂等的。

POST /tickets # 新建一个ticket

HTTP PUT方法表示更新资源，它会直接把实体数据替换到服务器中的资源，多次调用只会产生一次影响，所以是幂等的。

PUT /tickets/12 # 更新ticket 12

HTTP PATCH方法表示更新部分资源，它提供的实体需要根据程序或其他协议的定义，解析后在服务器上执行，以此来修改服务器上的资源。换句话说，PATCH请求是会执行某个程序的，如果重复提交，程序可能执行多次，对服务器上的资源可能造成额外的影响，所以是非幂等的。

PATCH /tickets/12 # 更新ticket 12

服务器对以上方法的处理是：当调用该方法更新部分字段时，会将这条ticket记录的操作记录加1，也就是说，每次调用都会改变资源状态，因此这是非幂等的操作。

HTTP DELETE方法表示删除资源，调用一次和多次对资源产生影响是相同的，所以是幂等的。

DELETE /tickets/12 # 删除ticekt 12

# 如何设计符合幂等性的高质量RESTful API

HTTP GET方法 vs HTTP POST方法：也许，你会想起一个面试题。HTTP请求的GET与POST方式有什么区别？你可能会回答到：GET方式通过URL提交数据，数据在URL中可以看到；POST方式，数据放置在HTML HEADER内提交。但是，我们现在从RESTful的资源角度来看待问题，HTTP GET方法是幂等的，所以它适合作为查询操作，HTTP POST方法是非幂等的，所以用来表示新增操作。

但是，也有例外，我们有的时候可能需要把查询方法改造成HTTP POST方法。比如，超长（1k）的GET URL使用POST方法来替代，因为GET受到URL长度的限制。虽然，它不符合幂等性，但是它是一种折中的方案。

HTTP POST方法 vs HTTP PUT方法：对于HTTP POST方法和HTTP PUT方法，我们一般的理解是POST表示创建资源，PUT表示更新资源。当然，这个是正确的理解。但是，实际上，两个方法都可以用于创建资源，更为本质的差别是在幂等性。HTTP POST方法是非幂等，所以用来表示创建资源，HTTP PUT方法是幂等的，因此表示更新资源更加贴切。

HTTP PUT方法 vs HTTP PATCH方法：HTTP PUT方法和HTTP PATCH方法，都是用来表述更新资源，它们之间有什么区别呢？我们一般的理解是PUT表示更新全部资源，PATCH表示更新部分资源。首先，这个是我们遵守的第一准则。根据上面的描述，PATCH方法是非幂等的，因此我们在设计我们服务端的RESTful API的时候，也需要考虑。如果，我们想要明确的告诉调用者我们的资源是幂等的，我的设计更倾向于使用HTTP PUT方法。

参考：

<http://www.cnblogs.com/weidagang2046/archive/2011/06/04/2063696.html#!comments>

<http://www.tuicool.com/articles/E7zeeii>

<http://blog.csdn.net/tjgamejx2/article/details/51011425>