## JS闭包

什么是闭包：闭包就是能够读取其他函数内部变量的函数。函数没有被释放，整条作用域链上的局部变量都将得到保留。因为js中只有函数内部的子函数才能读取局部变量，所以可以将闭包简单理解成定义在一个函数内部的函数。

关于javascript闭包中的this对象

this对象是运行时基于函数的执行环境绑定的：在全局函数中，this等于window，而当函数被作为某个对象的方法调用时，this等于那个对象。

匿名函数无法取得其包含作用域（外部作用域）的this对象，所以指向的是window,要想获得外部作用域的this对象，可以把外部作用域中的this保存在闭包可以访问的变量里，即包含匿名函数的函数

## 微信扫码登陆

当打开微信网页版时，用户不用输入用户名和密码，只需扫一个二维码即可实现登陆。本质上二维码只是一段文本，如何实现扫码登陆的呢。

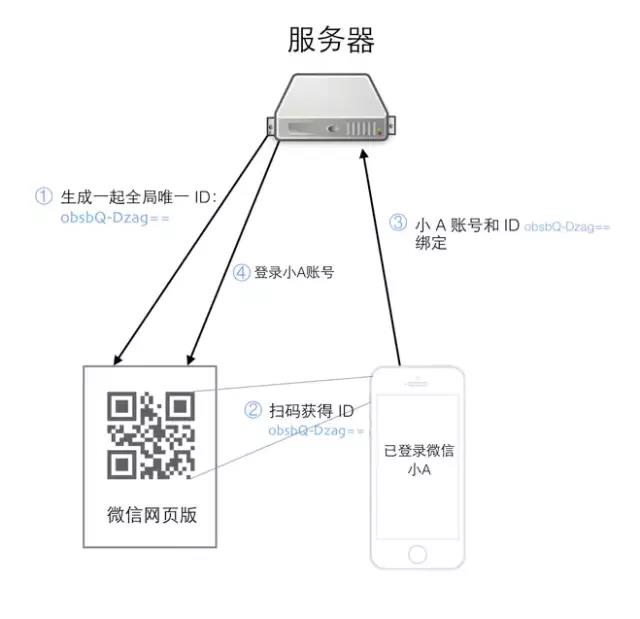
一下下是它的详细流程：

1：用户 A 访问微信网页版，微信服务器为这个会话生成一个全局唯一的 ID，图中 obsbQ-Dzag== 就是这个 ID，此时系统并不知道访问者是谁。

2：用户A打开自己的手机微信并扫描这个二维码，并提示用户是否确认登录。

3：手机上的微信是登录状态，用户点击确认登录后，手机上的微信客户端将微信账号和这个扫描得到的 ID 一起提交到服务器

4：服务器将这个 ID 和用户 A 的微信号绑定在一起，并通知网页版微信，这个 ID 对应的微信号为用户 A，网页版微信加载用户 A 的微信信息，至此，扫码登录全部流程完成



## 高性能网络编程之accept建立连接

众所周知，IO是计算机上最慢的部分，先不看磁盘IO，针对网络编程，自然是针对网络IO。网络协议对网络IO影响很大，当下，TCP/IP协议是毫无疑问的主流协议，本文就主要以TCP协议为例来说明网络IO。

网络IO中应用服务器往往聚焦于以下几个由网络IO组成的功能中：A）与客户端建立起TCP连接。B）读取客户端的请求流。C）向客户端发送响应流。D）关闭TCP连接。E）向其他服务器发起TCP连接。

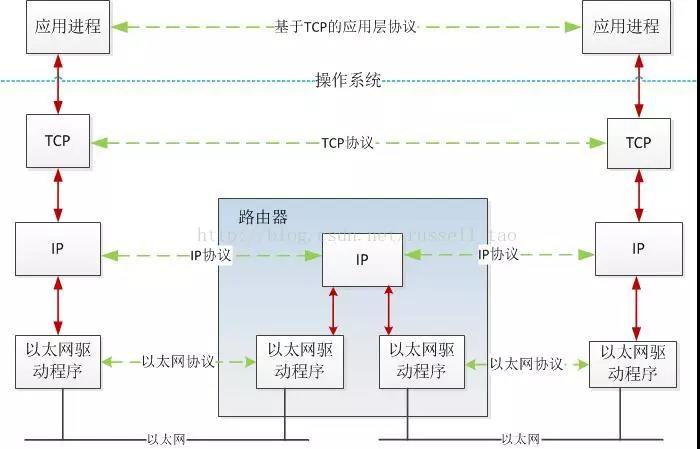
网络、协议、应用服务器间的关系：

ip网络层只专注于每一个网络分组如何到达目的主机，而不管目的主机如何处理。

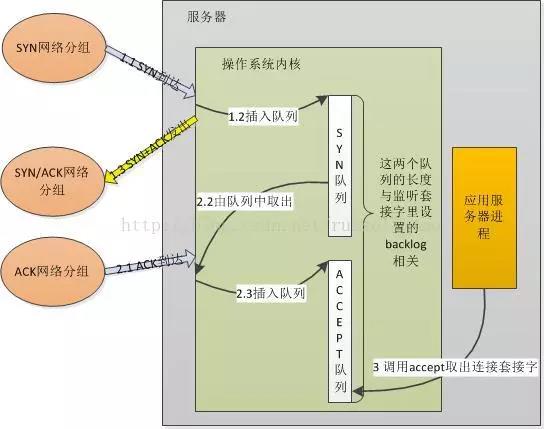
传输层最基本的功能是专注于端到端，也就是一台主机上的进程发出的包，如何到达目的主机上的某个进程。

TCP层为了可靠性，还额外需要解决3个大问题：丢包（网络分组在传输中存在的丢失）、重复（协议层异常引发的多个相同网络分组）、延迟（很久后网络分组才到达目的地）。

链路层则只关心以太网或其他二层网络内网络包的传输。



### 应用层，往往只需要调用类似于accept的API就可以建立TCP连接，那么应用层如何与accept交互呢



这两个队列是内核实现的，当服务器绑定、监听了某个端口后，这个端口的SYN队列和ACCEPT队列就建立好了。

客户端使用connect向服务器发起TCP连接，当图中1.1步骤客户端的SYN包到达了服务器后，内核会把这一信息放到SYN队列（即未完成握手队列）中，同时回一个SYN+ACK包给客户端。一段时间后，在图中2.1步骤中客户端再次发来了针对服务器SYN包的ACK网络分组时，内核会把连接从SYN队列中取出，再把这个连接放到ACCEPT队列（即已完成握手队列）中。而服务器在第3步调用accept时，其实就是直接从ACCEPT队列中取出已经建立成功的连接套接字而已。

现有我们可以来讨论应用层组件：为何有的应用服务器进程中，会单独使用1个线程，只调用accept方法来建立连接，例如tomcat；有的应用服务器进程中，却用1个线程做所有的事，包括accept获取新连接。

原因在于：首先，SYN队列和ACCEPT队列都不是无限长度的，它们的长度限制与调用listen监听某个地址端口时传递的backlog参数有关。既然队列长度是一个值，那么，队列会满吗？

答案是当然会，如果上图中第1步执行的速度大于第2步执行的速度，SYN队列就会不断增大直到队列满；如果第2步执行的速度远大于第3步执行的速度，ACCEPT队列同样会达到上限。第1、2步不是应用程序可控的，但第3步却是应用程序的行为，假设进程中调用accept获取新连接的代码段长期得不到执行，例如获取不到锁、IO阻塞等。

那么，这两个队列满了后，新的请求到达了又将发生什么？

若SYN队列满，则会直接丢弃请求，即新的SYN网络分组会被丢弃；如果ACCEPT队列满，则不会导致放弃连接，也不会把连接从SYN列队中移出，这会加剧SYN队列的增长。所以，对应用服务器来说，如果ACCEPT队列中有已经建立好的TCP连接，却没有及时的把它取出来，这样，一旦导致两个队列满了后，就会使客户端不能再建立新连接，引发严重问题。

所以，如TOMCAT等服务器会使用独立的线程，只做accept获取连接这一件事，以防止不能及时的去accept获取连接。

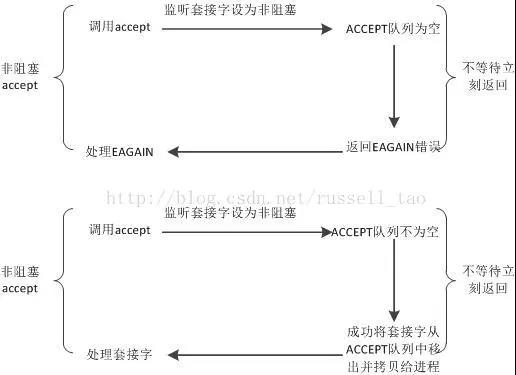
那么，为什么如Nginx等一些服务器，在一个线程内做accept的同时，还会做其他IO等操作呢？

这里就带出阻塞和非阻塞的概念。应用程序可以把listen时设置的套接字设为非阻塞模式（默认为阻塞模式），这两种模式会导致accept方法有不同的行为。对阻塞套接字，accept行为如下图：



这幅图中可以看到，阻塞套接字上使用accept，第一个阶段是等待ACCEPT队列不为空的阶段，它耗时不定，由客户端是否向自己发起了TCP请求而定，可能会耗时很长。

对非阻塞套接字，accept会有两种返回，如下图：



非阻塞套接字上的accept，不存在等待ACCEPT队列不为空的阶段，它要么返回成功并拿到建立好的连接，要么返回失败。

所以，企业级的服务器进程中，若某一线程既使用accept获取新连接，又继续在这个连接上读、写字符流，那么，这个连接对应的套接字通常要设为非阻塞。原因如上图，调用accept时不会长期占用所属线程的CPU时间片，使得线程能够及时的做其他工作。

## JAVA代理

通俗的来说，一个对象A要实现一个方法，在没代理前是由对象A做，代理之后由A的代理对象B来做

### 静态代理

在程序运行前就已经存在了代理类的字节码文件，代理类和原始类的关系在运行前就已经确定。

|  |
| --- |
| package test.staticProxy;  // 接口  public interface IUserDao {   void save();   void find();  }  //目标对象  class UserDao implements IUserDao{   @Override   public void save() {     System.out.println("模拟：保存用户！");   }   @Override   public void find() {     System.out.println("模拟：查询用户");   }  }  /\*\*     静态代理           特点：   1. 目标对象必须要实现接口   2. 代理对象，要实现与目标对象一样的接口  \*/  class UserDaoProxy implements IUserDao{   // 代理对象，需要维护一个目标对象   private IUserDao target = new UserDao();   @Override   public void save() {     System.out.println("代理操作： 开启事务...");     target.save();   // 执行目标对象的方法     System.out.println("代理操作：提交事务...");   }   @Override   public void find() {     target.find();   }  } |

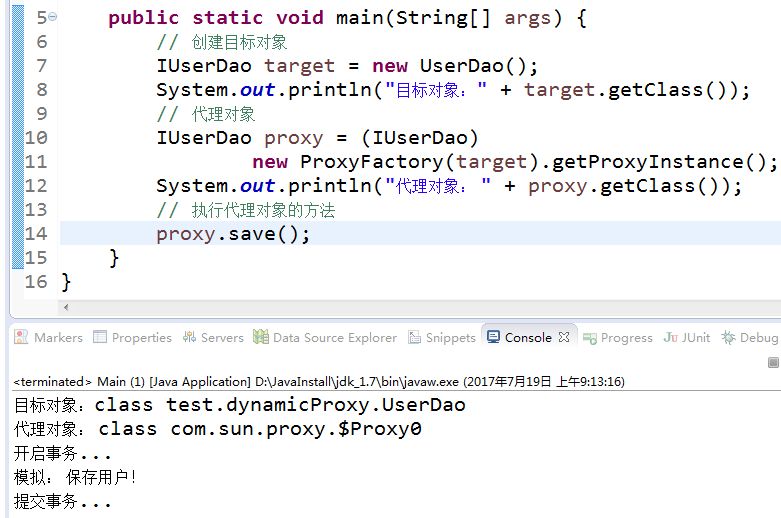
静态代理虽然保证了业务类只需关注逻辑本身，代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理。再者，如果增加一个方法，除了实现类需要实现这个方法外，所有的代理类也要实现此方法。增加了代码的维护成本。那么要如何解决呢?答案是使用动态代理。

### 动态代理模式

动态代理类的源码是在程序运行期间通过JVM反射等机制动态生成，代理类和委托类的关系是运行时才确定的。

|  |
| --- |
| package test.staticProxy;  package test.dynamicProxy;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  // 接口  public interface IUserDao {  void save();  void find();  }  //目标对象  class UserDao implements IUserDao{  @Override  public void save() {  System.out.println("模拟： 保存用户！");  }  @Override  public void find() {  System.out.println("查询");  }  }  /\*\*  \* 动态代理：  \* 代理工厂，给多个目标对象生成代理对象！  \*  \*/  class ProxyFactory {  // 接收一个目标对象  private Object target;  public ProxyFactory(Object target) {  this.target = target;  }  // 返回对目标对象(target)代理后的对象(proxy)  public Object getProxyInstance() {  Object proxy = Proxy.newProxyInstance(  target.getClass().getClassLoader(), // 目标对象使用的类加载器  target.getClass().getInterfaces(), // 目标对象实现的所有接口  new InvocationHandler() { // 执行代理对象方法时候触发  @Override  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {    // 获取当前执行的方法的方法名  String methodName = method.getName();  // 方法返回值  Object result = ;  if ("find".equals(methodName)) {  // 直接调用目标对象方法  result = method.invoke(target, args);  } else {  System.out.println("开启事务...");  // 执行目标对象方法  result = method.invoke(target, args);  System.out.println("提交事务...");  }  return result;  }  }  );  return proxy;  }  } |

测试结果如下



使用jdk生成的动态代理的前提是目标类必须有实现的接口。但这里又引入一个问题,如果某个类没有实现接口,就不能使用JDK动态代理,所以Cglib代理就是解决这个问题的。

### Cglib代理

Cglib是以动态生成的子类继承目标的方式实现，在运行期动态的在内存中构建一个子类。

Cglib使用的前提是目标类不能为final修饰。因为final修饰的类不能被继承。

通过定义和前面代码我们可以发现3点：

1.AOP是基于动态代理模式。

2.AOP是方法级别的（要测试的方法不能为static修饰，因为接口中不能存在静态方法，编译就会报错）。

3.AOP可以分离业务代码和关注点代码（重复代码），在执行业务代码时，动态的注入关注点代码。切面就是关注点代码形成的类。

### Spring AOP综合两种代理方式的使用

如果目标类没有实现接口，且class为final修饰的，则不能进行Spring AOP编程！

## 无状态

给定一个函数（方法）y=f(x)，对于一个给定的x，函数经过计算总是能得到相同的结果，不依赖外部的状态，那这个函数就是无状态的。

好处：因为函数的内部变量和数据在使用完之后就释放了，不存在对外部数据的操作，在多线程的并发上不会存任何问题。

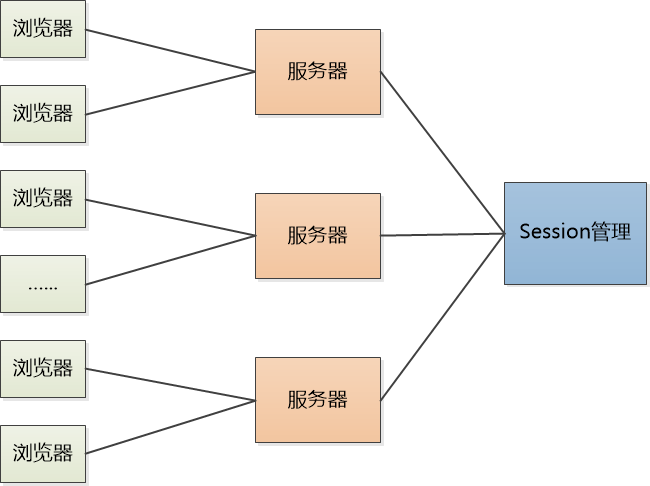
### http的无状态思想

HTTP每次访问一个静态HTML页面的时候，对于服务器来讲，是不是就相当于调用了一个函数，函数输入：一个URL路径， 函数输出：HTML页面。这个服务器也不会记录每次请求的是谁，只要执行这个'函数调用'就可以了。

好处：如果一个服务器访问量过大，可以轻松实现集群。

问题：在服务器端无状态确实是个美好的世界， 可是现实很残酷，没有状态不行啊，一个人登录了，我们得记住他是谁吧，他往购物车里加入商品，我们也得记下来吧。

解决方案：把状态（session）转移存储到另外一个地方，尽量服务器恢复到无状态的'y=f(x)'。



### 无状态对象（不可变对象）

一个对象没有实例变量，或者实例变量是final的(即不可变的实例变量)。

Controller, Service 默认都是单例，运行期只有一个实例，他们的方法应该是y=f(x)这样的无状态方法，轻易不要在里边放置共享的实例变量，要不然多线程并发操作（tomcat就是多线程处理请求）就可能出问题了。

但是如果确实需要共享的变量就把他放入threalocal

### ThreadLocal

ThreadLoal是每个线程保存私有变量的地方，因为多线程并发时操作公共变量容易导致变量被覆盖，所以可将这个公共变量存放在被使用线程的ThreadLoal中。

用例：拦截器拦截每一个用户信息，将用户的user-agent信息存放在当前线程的ThreadLoca中，在controller的层层调用中的某一层可能会用到用户的user-agent信息，然后直接获取就行，因为tomcat使用的是线程池处理用户请求，最后一定要删除存放的信息（原因是线程池的线程用完不会销毁，会变成阻塞状态等待下一个用户请求，里面存放的变量不会被删除）。

### 代码

LocalUser：获取和设置ThreadLocal

|  |
| --- |
| package com.example.demo.util;  /\*\*  \* Created by Administrator on 2018/11/26 0026.  \*/  public class LocalUser {  //处理每个线程的用户  private static ThreadLocal<String> USER\_LOCAL = new ThreadLocal<>();  public static void setCurrentUser(String user) {  USER\_LOCAL.set(user);  }  public static String gerCurrentUser(){  return USER\_LOCAL.get();  }  public static void removeUser(){  USER\_LOCAL.remove();  }  } |

UserInterceptor：拦截器

|  |
| --- |
| package com.example.demo.interceptor;  import com.example.demo.util.LocalUser;  import org.springframework.stereotype.Component;  import org.springframework.web.servlet.ModelAndView;  import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  /\*\*  \* Created by Administrator on 2018/11/26 0026.  \*/  @Component  public class UserInterceptor extends HandlerInterceptorAdapter {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  String user = request.getHeader("User-Agent");  LocalUser.setCurrentUser("i'm " + user);  return true;  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)  throws Exception {  LocalUser.removeUser();  }  } |

Controller

|  |
| --- |
| package com.example.demo.controller;  import com.example.demo.service.ShowUserAgentService;  import com.example.demo.util.LocalUser;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  /\*\*  \* Created by Administrator on 2018/11/26 0026.  \*/  @RestController  public class ShowUserController {  @Autowired  ShowUserAgentService showUserAgentService;  @RequestMapping("showUser")  public String showUser(){  String user = showUserAgentService.getUserAgent();  return user;  }  } |

Service

|  |
| --- |
| package com.example.demo.service.serviceImpl;  import com.example.demo.service.ShowUserAgentService;  import com.example.demo.util.LocalUser;  import org.springframework.stereotype.Service;  /\*\*  \* Created by Administrator on 2018/11/26 0026.  \*/  @Service  public class ShowUserAgentServiceImpl implements ShowUserAgentService {  @Override  public String getUserAgent() {  String userAgent = LocalUser.gerCurrentUser();  return userAgent;  }  } |

## 分布式事务

分布式系统涉及到了跨数据库操作，如果一个事务需要跨数据库，如何保证事务的ACID。

### JTA（Java Transaction API）

JAVA提供了JTA负责协调各个数据库的事务提交，可以保证强一致性。 为了实现分布式事务, JTA特设两个阶段

阶段一：全局的事务管理器向各个数据库发出准备消息。 各个数据库需要在本地把一切都准备好，执行操作，锁住资源， 记录redo/undo 日志， 但是并不提交， 总而言之，要进入一个时刻准备提交或回滚的状态， 然后向全局事务管理器报告是否准备好了。

阶段2： 如果所有的数据库都报告说准备好了， 那全局的事务管理器就下命令： 提交， 这时候各个数据库才真正提交 ， 由于之前已经万事具备，只欠东风，只需要快速完成本地提交即可。

如果有任何一个数据库报告说没准备好， 事务管理器就下命令： 放弃， 这时候各个数据库要执行回滚操作， 并且释放各种在阶段1锁住的资源。

**弊端**：JTA为了保证强一致性付出的代价太大，不利于高并发系统的运作。分布式系统很多情况下只要保证最终一致性就行，不需要保证强一致性，比如A给B转账，不必要求A的账户里少了一百，B的账户就立即多出一百，可以一段时间后再多出那一百。

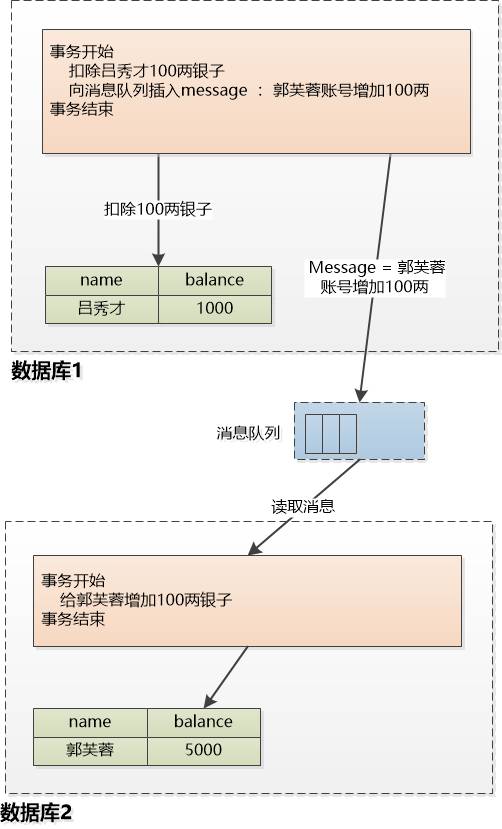
### 消息队列

假设两个账户（吕秀才和郭芙蓉）在两个独立的数据库中， 原来设计的JTA是要求从吕秀才账户减去100两银子， 然后在郭芙蓉账户加上100两银子， 这两个操作要么全部做完，要么全部不做， 但是在网络的环境下， 这是不大容易做到的， 或者说在高并发的情况下做到的代价太高。

为了保证分布式系统的最终一致性，可以使用消息队列。

#### 使用方案一：

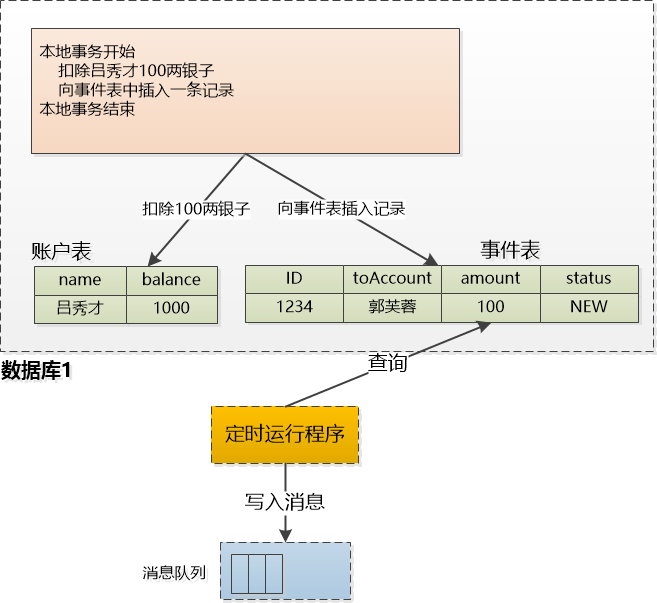
利用消息队列可以这样做：想从吕秀才账户转100两银子给郭芙蓉， 需要在数据库1发起一个事务， 从吕秀才账户扣除100两， 然后还得向消息队列插入一条给郭芙蓉账号增加100两的消息（**消息队列的数据都是持久化到硬盘上的， 不会丢失，所以宕机了也不怕**）， 然后这个数据库1的事务就结束了！ 消息队列中的消息会在某一刻被读取出来，进行处理，给郭芙蓉的账号增加100两。



**弊端：**事务1（从吕秀才账户扣除100两， 然后向消息队列插入一条给郭芙蓉账号增加100两的消息），操作的是两个完全不同的东西，无法保证事务的原子性。

#### 改进方案：

可以添加一个‘事件表’， 转账开始的时候，把吕秀才的100两银子扣除， 同时还向事件表插入一行记录： 需要向郭芙蓉转100两， 由于这两个表是在同一个数据库中，所以直接使用本地事务就行。不用什么分布式事务。再添加一个定时器程序，从事件表中取出记录， 向MQ写入消息， 然后把记录的状态改成‘DONE’， 这样下次就不用再去取去处理了。（定时运行的程序可以出错，可以向消息队列写入多次 ‘给郭芙蓉账号增加100两银子’ 这样的消息， 但是郭芙蓉那边在执行的时候， 肯定也要判断之前是否执行过了， 如果没有的话就增加， 如果执行过了就简单的抛弃这个消息即可）



## 第三方授权

一个应用要访问另外一个应用的资源， 该如何授权？

比如：

1. 微信公众平台的网站要通过微信获取关注者的个人信息
2. 一个信用卡管家的网站要读取用户网易邮箱的信息

### Resource Owner Password Credentials Grant(资源所有者密码凭据许可)

信用卡管家网站读取用户网易邮箱信息，需要用户提供用户名和密码，程序通过用户名和密码登陆访问。

问题：十分不安全。用户信息容易被窃取

### Implicit Grant(隐式许可)

1. 信用卡管家网站向网易申请授权，网易给网站提供app\_id 和app\_secret。
2. 为用户提供一个入口，用户点击之后会重定向到网易的认证系统去登陆（当然要携带网易提供的app\_id和app\_secret，网易才知道你的网站已经被授权），认证系统会让你输入用户名和密码并提示你是否允许该网站访问你的邮箱信息，确认以后会再次重定向到信用卡管家网站，同时带一个“token”过来。
3. 用这个“token”访问网易的API即可获得用户的信息

注意事项：这个token是用hash mark（#）进行标识返回的，表示它是一个fragment。

Fragment的特性：

1，# 有别于 ?，? 后面的查询字符串会被网络请求带上服务器，而 fragment 不会被发送的服务器；

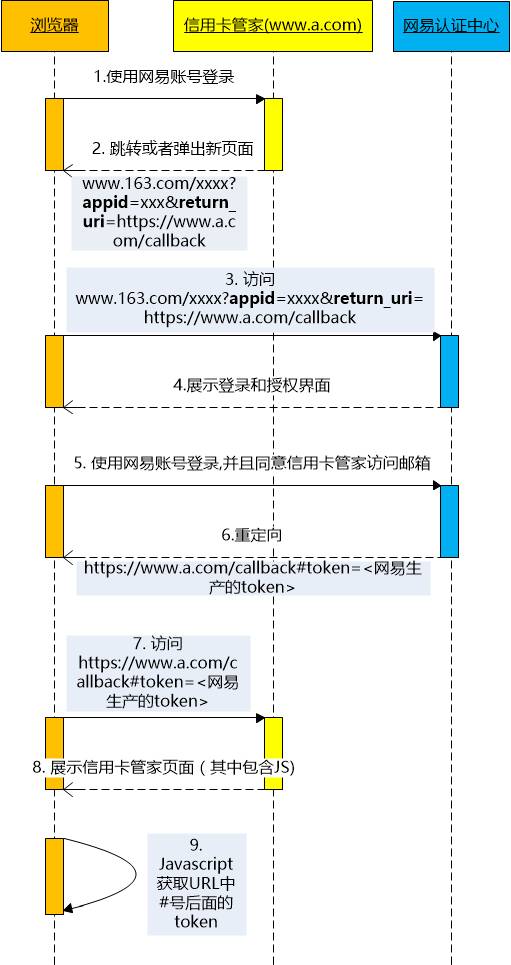
2，fragment 的改变不会触发浏览器刷新页面，但是会生成浏览历史；

3，fragment 会被浏览器根据文件媒体类型（MIME type）进行对应的处理；

1. Google 的搜索引擎会忽略 # 及其后面的字符串。

使用Fragment的目的：只有Javascript 能访问它，并且它不会再次通过http request 发到别的服务器， 这是为了提高安全性

信用卡管家要获取用户网易邮箱的信息为例展示隐私许可步骤



问题：这个token 以明文的方式发送给了用户的浏览器， 虽然是https ，不会被别人窃取，可是浏览器的历史记录或者访问日志中就能找到还是存在安全问题

### Authorization Code Grant(授权码许可)

和之前的思路类似，只是引用了一个授权码(authorization code)，当用户用网易账号登录的时候， 网易认证中心这一次不直接发token,而是发一个授权码(authorization code)，信用卡管家服务器端获得authorization code，在后台再次访问网易认证中心，这次才能获得真正的token。

注意：授权码和信用卡管家申请的app\_id，app\_secret关联， 只有信用卡管家发出的token请求， 网易认证中心才认为合法； 为了安全性，还可以让授权码有时间限制，比如5分钟失效，还有可以让授权码只能换一次token, 第二次就不行了。

