目录

[web3j以太坊开发详解 3](#_Toc9120698)

[以太坊概述 3](#_Toc9120699)

[智能合约平台 3](#_Toc9120700)

[JSON RPC与web3j 4](#_Toc9120701)

[web3j体系概述 5](#_Toc9120702)

[课程内容概述 5](#_Toc9120703)

[第二章 hello,web3j 6](#_Toc9120704)

[第三章 账户管理 6](#_Toc9120705)

[第四章 状态与交易 6](#_Toc9120706)

[第五章 智能合约 6](#_Toc9120707)

[第六章 过滤器与事件 6](#_Toc9120708)

[hello,web3j 7](#_Toc9120709)

[Ganache概述 7](#_Toc9120710)

[使用ganache 8](#_Toc9120711)

[引入项目依赖 9](#_Toc9120712)

[创建Web3j对象 12](#_Toc9120713)

[与以太坊节点交互 13](#_Toc9120714)

[构建运行 14](#_Toc9120715)

[调整日志输出等级 15](#_Toc9120716)

[账户管理 16](#_Toc9120717)

[账户管理概述 16](#_Toc9120718)

[私钥、公钥与账户 17](#_Toc9120719)

[创建账户 17](#_Toc9120720)

[导入私钥 18](#_Toc9120721)

[使用钱包 19](#_Toc9120722)

[账户凭证 21](#_Toc9120723)

[节点管理的账户 22](#_Toc9120724)

[状态与交易 23](#_Toc9120725)

[概述 23](#_Toc9120726)

[状态与激励 24](#_Toc9120727)

[以太坊状态机 26](#_Toc9120728)

[获取账户余额 27](#_Toc9120729)

[以太坊货币单位 28](#_Toc9120730)

[货币单位表示与换算 30](#_Toc9120731)

[交易类型 31](#_Toc9120732)

[提交普通交易 32](#_Toc9120733)

[获取交易收据 34](#_Toc9120734)

[gas价格与用量 35](#_Toc9120735)

[web3j中为交易设置gas参数 36](#_Toc9120736)

[使用裸交易 37](#_Toc9120737)

[受控交易与交易管理器 39](#_Toc9120738)

[裸交易管理器 41](#_Toc9120739)

[智能合约 42](#_Toc9120740)

[概述 42](#_Toc9120741)

[去中心化投票 43](#_Toc9120742)

[设计合约接口 45](#_Toc9120743)

[编写合约代码 45](#_Toc9120744)

[生成合约包装类 47](#_Toc9120745)

[编译合约 48](#_Toc9120746)

[生成包装类 48](#_Toc9120747)

[编写合约处理脚本 49](#_Toc9120748)

[部署/载入合约 50](#_Toc9120749)

[载入已部署合约 52](#_Toc9120750)

[调用合约方法 52](#_Toc9120751)

[使用不同的账户 53](#_Toc9120752)

[过滤器与事件 53](#_Toc9120753)

[通知机制概述 53](#_Toc9120754)

[监听新块事件 55](#_Toc9120755)

[设置轮询周期 56](#_Toc9120756)

[监听新交易事件 56](#_Toc9120757)

[监听待定交易事件 58](#_Toc9120758)

[合约事件 59](#_Toc9120759)

[监听合约事件 60](#_Toc9120760)

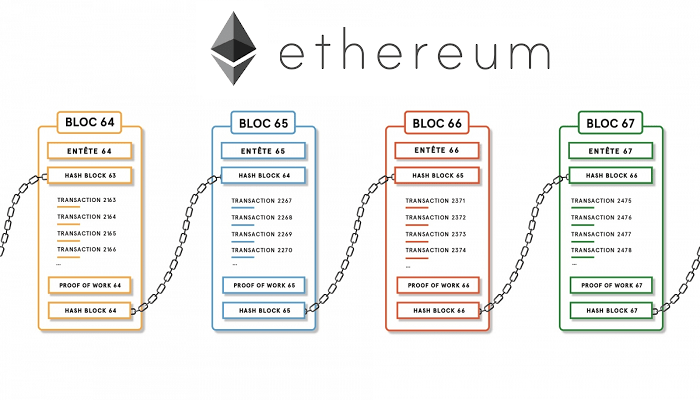
web3j以太坊开发详解

***汇智网 / hubwiz.com / 2018-4-1***

本课程详细讲解如何使用web3j为Java应用或Android App增加以太坊区块链支持，课程内容即涉及以太坊中的核心概念，例如账户管理、状态与交易、智能合约开发与交互、过滤器和事件等，同时也详细说明如何使用web3j提供的开发接口与以太坊进行交互，是java工程师学习以太坊应用开发的不二选择

## 以太坊概述

以太坊是备受关注的区块链，它基于密码学技术和P2P通信技术 构建了一个去中心化的平台，所有的交易同步保存在每个节点中， 通过将区块单向级联成链，以太坊有效的保证了交易的不可篡改：



## 智能合约平台

以太坊是第一个实现了虚拟机的区块链，因此为智能合约 - Smart Contract - 的运行提供了良好的支持环境。也正因为这个原因，以太坊被称为区块链 2.0，以区别于比特币代表的以数字加密货币为核心特征的区块链1.0。

可以将智能合约理解为机器之间的合同约定，在满足一定条件时自动 执行约定好的逻辑，例如在保险理赔流程中，如果理赔条件满足就自动 将赔偿金释放给出险人，这个流程就可以使用智能合约来实现。

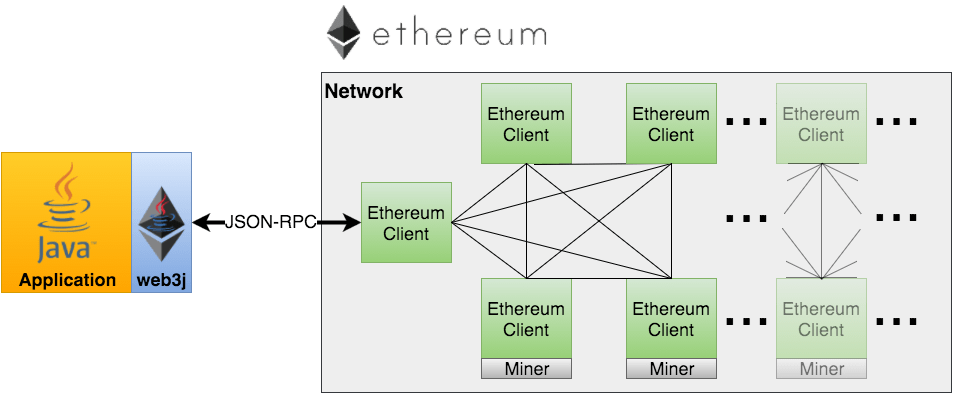
有多种语言可以开发以太坊智能合约，但目前最常用的是类似于JavaScript的 Solidity语言。本课程中将采用Solidity讲解智能合约的开发。

阅读教程回答问题：为什么说以太坊是区块链2.0？

## JSON RPC与web3j

如果我们希望构造一个去中心化应用（DApp），除了智能合约的开发， 通常还需要使用其他开发语言为用户提供操作智能合约的用户接口，例如 开发一个网页、一个手机App或者一个桌面应用。这些代码都需要与以太坊 进行交互。

以太坊规定了每个节点需要实现的[JSON RPC API](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/) 应用开发接口，该接口是传输无关的，应用程序可以通过HTTP、websocket或IPC等多种 通信机制来使用该接口协议操作以太坊节点：



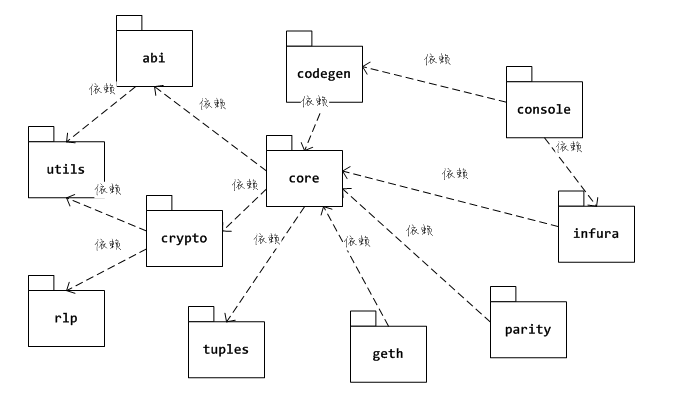
理论上你可以使用任何语言基于JSON RPC接口开发出以太坊之上的 去中心化应用，不过为了提高开发效率，更好的办法是 使用特定语言的JSON RPC封装库，这些库封装了JSON RPC的协议细节， 有助于开发人员聚焦在业务逻辑的实现上。

web3j是一个轻量级的用于集成以太坊功能的Java开发库，它是Java版本的以太坊JSON RPC 接口协议封装实现，如果需要将你的Java应用或Android应用接入以太坊，用web3j就对了。

阅读教程回答问题：web3j与以太坊JSON RPC API是什么关系？

## web3j体系概述

web3j的功能组织在不同的包中，下图展示了org.web3j主要包之间的依赖关系：



**core**：JSON RPC协议的封装主要由包org.web3j.core实现，它依赖于org.web3j.crypto包提供 的密钥与签名相关的功能，以及org.web3j.abi包提供的java/solidity类型映射支持。

**console**：org.web3j.console包实现了一个可以单独运行的命令行程序web3j，我们将使用它来 生成solidity合约的Java封装类，其中，org.web3j.codegen包实现了从abi到java封装类的代码生成。

**节点相关**：org.web3j.infura包封装了对Infura公共节点旳http访问服务接口，org.web3j.geth和org.web3j.parity则分别封装了这两种常用以太坊节点软件旳管理接口。

阅读教程回答问题：web3j的哪个包实现了以太坊JSON RPC接口协议的封装？

## 课程内容概述

本课程的目的是帮助java工程师快速掌握使用web3j开发以太坊应用的技能，因此 主要以web3j的开发接口为主线来展开课程内容，同时穿插讲解以太坊的一些基本 概念，例如：账户、交易和智能合约的开发等。

### 第二章 hello,web3j

这一章将通过一个简单的java应用的开发来讲解使用 web3j进行以太坊应用开发的最简流程，通过这一部分的学习，你就可以在自己 的java应用中引入以太坊支持了。

### 第三章 账户管理

这一章将详细介绍web3j提供的账户管理接口。如果你 对开发钱包应用（中心化/去中心化）感兴趣，这部分内容会有很大的帮助。

### 第四章 状态与交易

这一章主要讲解web3j提供的交易操作接口，同时也介绍一些 重要的概念，例如状态、裸交易、gas等。这部分内容将帮助你理清java 应用与以太坊交互的大多数问题。

### 第五章 智能合约

这一章将通过一个投票合约的开发、编译、代码生成、部署与交互 的完整流程，讲解使用web3j操作solitiy合约的方法。

### 第六章 过滤器与事件

这一章主要讲解以太坊的通知机制和web3j的响应式封装接口。

课程为每个知识点都提供了相应的预置代码，你可以在实验环境的~/repo 目录下查看。

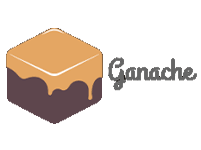
# hello,web3j

## Ganache概述

在这一部分，我们将开发一个最简单的Java控制台应用，来接入以太坊节点，并打印 所连接节点旳版本信息。通过这一部分的学习，你将掌握以下技能：

* 如何使用节点仿真器
* 如何创建一个Web3j对象
* 如何声明要连接的节点
* 如果向节点发送一个RPC调用并提取返回结果

我们将使用ganache来模拟以太坊节点。ganache虽然不是一个真正的以太坊节点软件， 但它完整实现了以太坊的JSON RPC接口，非常适合以太坊智能合约与去中心化应用开发的 学习与快速验证：



ganache启动后将在8545端口监听http请求，因此，我们将使用web3j将JSON RPC调用请求 使用http协议发送到节点旳8545端口。不同的节点软件可能会使用不同的监听端口，但 大部分节点软件通常默认使用8545端口。

以太坊规定了节点必须实现[web3\_clientVersion](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/1/1/) 调用来返回节点软件的版本信息，因此我们可以使用web3j对这个命令的封装来测试与 节点旳链接是否成功。

|  |
| --- |
| Ganache在window环境中的安装： |
| 1、安装nodejs，下载地址：<https://nodejs.org/en/download/> 中下载zip版本。参考安装步骤参考：参考：<https://www.cnblogs.com/hackyo/p/8110951.html> |
| 2、安装Ganache。参考地址：<http://www.blockchainbrother.com/article/2137>  2.1 安装命令：npm install -g ganache-cli  2.2 启动命令：anache-cli  2.3 启动命令选项参考：<https://my.oschina.net/u/3790537/blog/1802985/> |

## 使用ganache

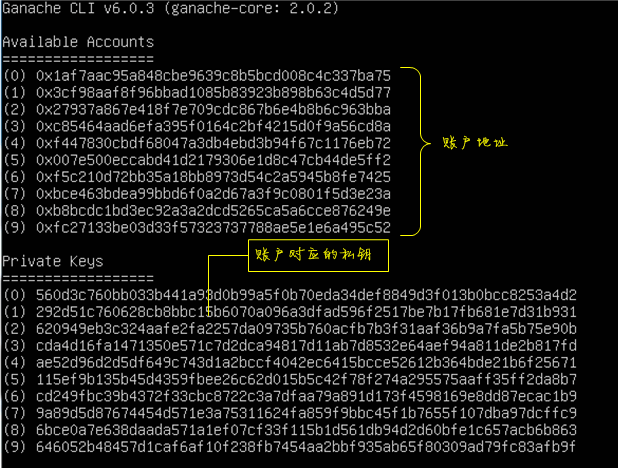
ganache-cli是以太坊节点仿真器软件ganache的命令行版本，可以方便开发者快速进行 以太坊DApp的开发与测试。在windows下你也可以使用其GUI版本。

启动ganache很简单，只需要在命令行执行ganache-cli即可：

~$ ganache-cli

ganache-cli是一个完整的词，-两边是没有空格的。

一切顺利的话，你会看到与下图类似的屏幕输出：



默认情况下，ganache会随机创建10个账户，每个账户中都有100ETH的余额。你可以在 命令行中指定一些参数来调整这一默认行为。例如使用-a或--acounts参数来指定 要创建的账户数量为20：

~$ ganache-cli -a 20

关于可选参数的详细信息，请参考[ganache-cli命令行参数说明](http://blog.hubwiz.com/2018/04/16/ganache-cli-manual/)。

## 引入项目依赖

为了避免被依赖关系绕晕，推荐你使用maven或gradle来开发Java应用。在 本课程中，我们将使用gradle构建工具。

首先创建项目源代码目录，然后创建gradle构建配置文件:

~$ mkdir -p hello/src/main/com/hubwiz/demo

~$ touch hello/src/main/com/huwiz/demo/App.java

~$ touch hello/build.gradle

在build.gradle中，我们使用java插件来管理基本的构件任务， 使用application插件来管理应用的运行：

gourp = 'com.hubwiz'

version = '0.0.1'

apply plugin: 'java'

sourceCompatibility = 1.8

apply plugin: 'application'

mainClassName = "com.hubwiz.demo.App"

然后声明要使用的仓库：

repositories {

mavenCentral()

}

最后声明我们的依赖项：

**dependencies** {

compile 'org.web3j:core:3.3.0',

'ch.qos.logback:logback-core:1.2.3',

'ch.qos.logback:logback-classic:1.2.3'

}

如果你是开发android应用，使用对应的包即可：

**dependencies** {

compile 'org.web3j:core:3.3.0-android',

'ch.qos.logback:logback-core:1.2.3',

'ch.qos.logback:logback-classic:1.2.3'

}

参考教程和示例代码：

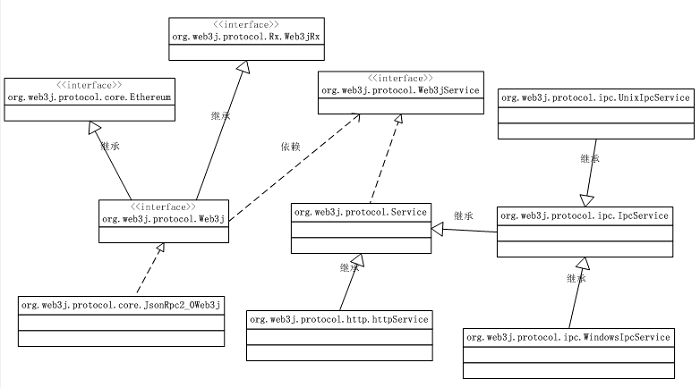
* 在~/repo目录下创建hello项目
* 在项目的gradle配置文件中启用java插件和application插件
* 在gradle配置文件中声明对web3j和logback的依赖

本章节改用maven构建开发java程序：

|  |
| --- |
| <!--  <dependency>  <groupId>org.web3j</groupId>  <artifactId>core</artifactId>  <version>3.3.0</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.2.3</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.2.3</version>  </dependency>  -->  <dependency>  <groupId>org.web3j</groupId>  <artifactId>core</artifactId>  <version>3.4.0</version>  </dependency> |
| **Java 8:**  <dependency>  <groupId>org.web3j</groupId>  <artifactId>core</artifactId>  <version>3.4.0</version>  </dependency>  **Android:**  <dependency>  <groupId>org.web3j</groupId>  <artifactId>core</artifactId>  <version>3.3.1-android</version>  </dependency> |
| **运行如下代码：**  import org.web3j.protocol.Web3j;  import org.web3j.protocol.http.HttpService;  import org.web3j.protocol.core.Request;  import org.web3j.protocol.core.methods.request.\*;  import org.web3j.protocol.core.methods.response.\*;  public class Hello {  public void run(){  try{  Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"));  Request<?,Web3ClientVersion> request = web3j.web3ClientVersion();  Web3ClientVersion web3ClientVersion = request.send();  String clientVersion = web3ClientVersion.getWeb3ClientVersion();  System.out.println(clientVersion);  }catch(Exception e){  System.out.print(e);  }  }  public static void main(String[] args) {  new Hello().run();  }  } |

## 创建Web3j对象

接口Web3j声明了以太坊JSON RPC相关的全部接口，该接口提供了 静态方法build()来返回一个该接口实现类JsonRpc2\_0Web3j的实例对象：



例如，下面的代码创建一个Web3j实例对象，该对象将后续的RPC调用通过HTTP 发送到本机运行的节点：

Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"));

使用HTTP服务既可以访问本机节点，也可以访问远程节点，因此，在大多数情况下 这是首选的传输服务。

不过web3j也提供了IPC访问实现。在Linux和windows下，可以分别使用UnixIpcService 和WindowsIpcService类来建立连接，两者分别是使用unix socket和named pipe 实现的。如果你的应用运行环境总是使用本地节点，可以使用IPC连接服务。

例如，下面的代码创建的web3j实例，其后续RPC调用将通过IPC发送给到本机的geth：

Web3j web3j = Web3j.build(new UnixIpcServer("/tmp/geth.ipc"))

注意：需要使用--rpcpath /tmp/geth.ipc选项来运行geth。

参考教程，在App.java中的main函数中创建web3j实例，并声明节点url为[http://localhost:8545](http://localhost:8545/)

## 与以太坊节点交互

一旦获得了Web3j实例，就可以调用其方法来与以太坊节点进行交互了。

例如，下面的代码读取所连接节点旳版本信息并在控制台输出：

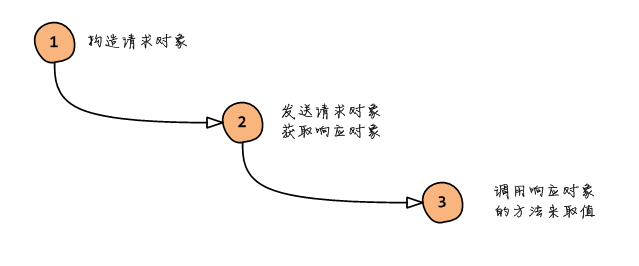
Request**<?**,Web3ClientVersion> request = web3j.web3ClientVersion();

Web3ClientVersion response = request.send();

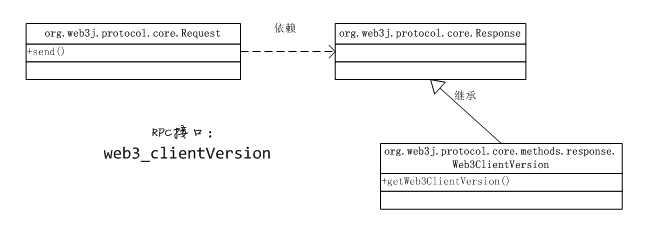
String clientVersion = response.getWeb3ClientVersion();

System.out.println("Client version: " + clientVersion);

上面的分步代码清晰展示了与以太坊节点交互的三个步骤：



web3j为[以太坊JSON RPC](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/) 中约定的每一种交互都定义了相应的请求类和响应类：



例如，对于[web3\_clientVersion](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/1/1/) 这个RPC接口，在web3j中对应的响应类就是Web3ClientVersion，而请求类则是 Request<?,Web3ClientVersion> ——使用泛型来约定参数类型和响应类型，以便进行自动转换。

通常情况下，我们可以采用链式写法，以避免这些中间类型，例如：

String clientVersion = web3j.web3ClientVersion().send().getWeb3ClientVersion();

参考教程，为App.java中的main函数实现打印节点版本信息的功能。

## 构建运行

在项目根目录下执行gradle run，即可直接编译运行应用：

~/repo/hello$ gradle run

也可以先执行构建，然后运行生成的jar：

~/repo/hello$ gradle build

~/repo/hello$ java -jar build/libs/c2-0.0.1.jar

如果一切正常，可以看到节点返回的版本信息：



如果你忘记了启动ganache，将得到如下信息：

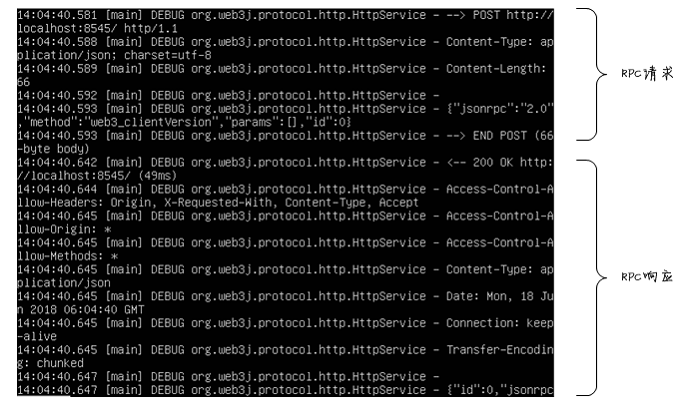
java.net.ConnectException: Failed to connect to localhost/127.0.0.1:8545

参考教程：

* 启动ganache
* 构建并运行hello应用

## 调整日志输出等级

web3j使用logback管理日志。默认情况下，web3j的日志运行在DEBUG级别，这意味着 每一次RPC请求与响应都会输出显示：



我们可以在项目资源中创建logback配置文件，然后设置web3j的日志等级为INFO。

首先创建logback.xml：

~/repo/hello$ mkdir -p src/main/resources

~/repo/hello$ touch src/main/resources/logback.xml

在logback.xml中设置org.web3j.protocol的日志输出级别为INFO：

**<?**xml version="1.0" encoding="UTF-8"**?>**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<logger name="org.web3j.protocol" level="DEBUG"/>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

重新运行应用，即可看到不再输出协议信息了。

参考教程，配置web3j的日志输出级别为INFO。

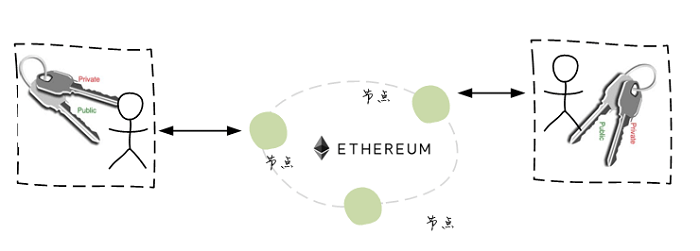
# 账户管理

## 账户管理概述

在这一部分，我们将学习如何使用web3j管理以太坊账户，这包括：

* 了解私钥、公钥和账户的关系
* 离线创建以太坊账户
* 导入其他账户私钥
* 创建和使用钱包
* 创建和使用账户凭证
* 读取节点管理的账户列表

以太坊作为一个去中心化的系统，必然不会采用中心化的账户管理 方案 —— 没有一个中心数据库来保存以太坊平台上的所有账户信息。 事实上，以太坊使用非对称密钥技术来进行身份识别，一个以太坊 账户对应着一对密钥：



web3j中账户管理的功能主要由org.web3j.crypto包实现，常用的类 包括：

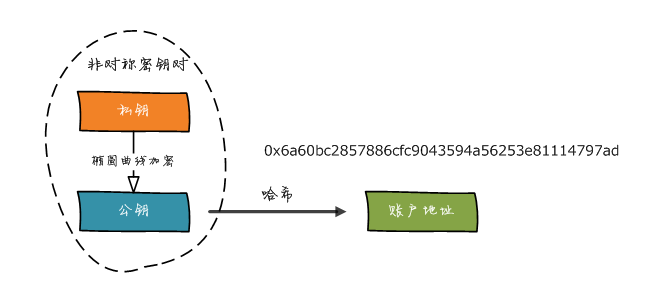
* ECKeyPair：表征密钥对的类
* WalletUtils：钱包辅助类
* Credentials：账户凭证类

可以使用ECKeyPair来创建新的密钥对，或者导入已有的私钥； WalletUtils可以用来生成或载入钱包文件，而Credentials则 包含了一个账户的全部标识信息，是执行交易的主体。

## 私钥、公钥与账户

以太坊使用非对称密钥对来进行身份识别，每一个账户都有 对应的私钥和公钥 —— 私钥用来签名、公钥则用来验证签名 —— 从而 在非可信的去中心化环境中实现身份验证。

事实上，在以太坊上账户仅仅是对应于特定非对称密钥对中公钥的20字节 哈希：



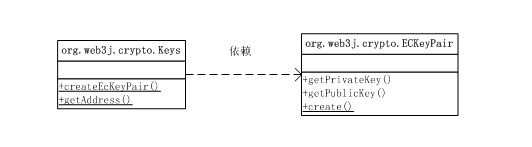
从私钥可以得到公钥，然后进一步得到账户地址，而反之则无效。 显然，以太坊不需要一个中心化的账户管理系统，我们可以根据以太坊约定 的算法自由地生成账户。

阅读教程回答问题：单独使用以下哪一项可以完全恢复一个以太坊账户：

* 私钥
* 公钥
* 账户地址

## 创建账户

在web3j中，使用org.web3j.crypto.Keys类来创建账户：



例如，下面的代码创建一个密钥对，然后生成账户地址。注意Keys 类的createEcKeyPair()和getAddress()都是静态方法：

ECKeyPair keyPair = Keys.createEcKeyPair();

String privateKey = keyPair.getPrivateKey().toString(16);

String publicKey = keyPair.getPublicKey().toString(16);

String account = Keys.getAddress(keyPair);

getAddress()方法有多个重载版本：

* public static String getAddress(ECKeyPair ecKeyPair)
* public static String getAddress(BigInteger publicKey)
* public static String getAddress(String publicKey)
* public static byte[] getAddress(byte[] publicKey)

显然，只需要公钥就可以获得对应的账户地址了。例如：

String publicKey = keyPair.getPublicKey().toString(16);

String account = Keys.getAddress(publicKey);

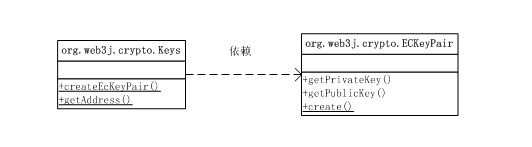
参考教程，在App.java中实现如下功能：

* 创建一个新的密钥对
* 在控制台输出该密钥对的私钥、公钥
* 在控制台输出该密钥对所对应的账户地址

## 导入私钥

我们已经知道，只有私钥是最关键的，公钥和账户都可以从私钥一步步 推导出来。

假如你之前已经通过其他方式有了一个账户，例如使用Metamask创建 的钱包，那么可以把该账户导入web3j。



使用org.web3j.crypto.ECKeyPair类的静态方法create()来导入 私钥。该方法有多个重载：

* public static ECKeyPair create(KeyPair keyPair)
* public static ECKeyPair create(BigInteger privateKey)
* public static ECKeyPair create(byte[] privateKey)

例如：

BigInteger privateKey = new BigInteger("133be114715e5fe528a1b8adf36792160601a2d63ab59d1fd454275b31328791",16);

ECKeyPair keyPair = ECKeyPair.create(privateKey);

String account = Keys.getAddress(keyPair);

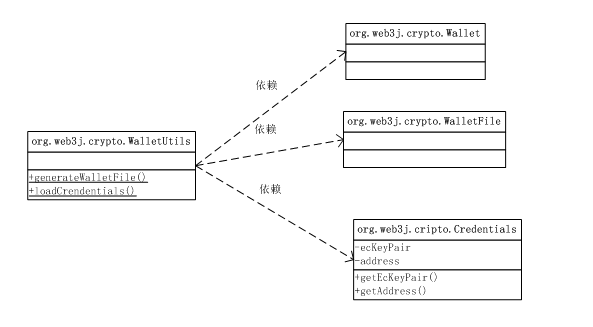
参考教程，使用以下私钥恢复公钥和账户地址：

0x133be114715e5fe528a1b8adf36792160601a2d63ab59d1fd454275b31328791

## 使用钱包

在web3j中，钱包这个概念对应的是加密的私钥文件。因此，在一个钱包中， 只能有一个私钥 —— 如果你需要在web3j中管理多个账户，就需要创建多个钱包 —— 这有别于我们通常提到的钱包应用的概念。

使用org.web3j.crypto.WalletUtils类来管理钱包文件的创建和加载：



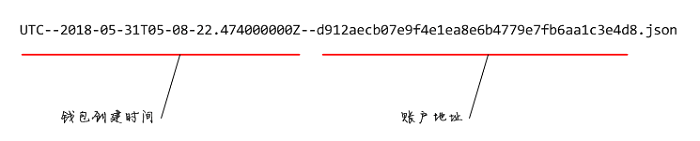
例如，下面的代码将在使用密码123在./keystore目录下创建一个新的钱包文件：

String pass = "123";

File destDir = new File("./keystore");

String fn = WalletUtils.generateWalletFile(pass,keyPair,destDir,true);

钱包文件名中包含了创建时间和对应的账户地址：



除了generateWalletFile()，WalletUtils中也提供了一些简化的方法：

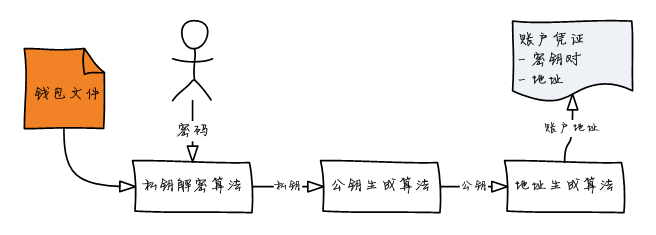
* public static String generateWalletFile(String password, ECKeyPair ecKeyPair, File destinationDirectory, boolean useFullScrypt)
* public static String generateNewWalletFile(String password, File destinationDirectory, boolean useFullScrypt)
* public static String generateLightNewWalletFile(String password, File destinationDirectory)
* public static String generateFullNewWalletFile(String password, File destinationDirectory)

参考教程，在App.java中实现如下功能：

* 使用密码abcd在目录./keystore下创建一个新的钱包
* 显示输出钱包的文件名，并指出该钱包的账户地址

## 账户凭证

钱包文件中保存的是加密后的私钥，显然，只要持有生成钱包文件时的密码， 就可以恢复私钥，进而重新获得公钥和账户地址：



下面的代码则使用WalletUtils类的loadCredentials()静态方法载入指定的钱包文件， 该方法返回一个Credentials凭证对象，此对象中包含了密钥对和账户地址信息：

Credentials credentials = WalletUtils.loadCredentials(pass,

"./keystore/UTC--2018-05-31T05-08-22.474000000Z--d912aecb07e9f4e1ea8e6b4779e7fb6aa1c3e4d8.json");

ECKeyPair keyPair = credentials.getEcKeyPair();

String account = credentials.getAddress();

凭证Credentials非常重要，因为在我们向以太坊提交交易时，总需要使用私钥 进行签名，而在Web3j中，通常使用凭证对象传入执行签名的方法。

除了通过载入钱包文件来实例化，也可以直接构建凭证对象。Credentials类提供 了静态方法create()的几种重载形式，可以使用私钥或密钥对来实例化凭证对象：

* public static Credentials create(ECKeyPair ecKeyPair)
* public static Credentials create(String privateKey, String publicKey)
* public static Credentials create(String privateKey)

参考教程，在App.java中实现以下功能：

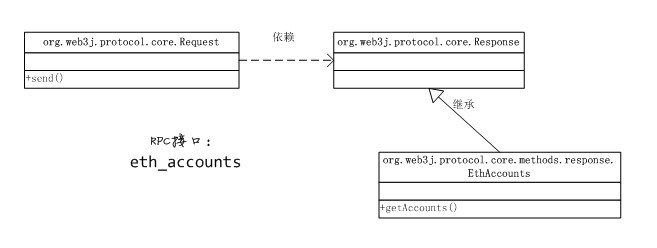
* 载入前一节创建的钱包文件
* 在控制台输出该钱包的私钥、公钥和账户地址

## 节点管理的账户

除了在web3j中离线创建账户，也可以使用节点管理账户。例如，geth、parity 等节点软件都提供了创建账户的用户接口和开发接口，ganache则会自动创建了一些 可供测试用的账户。

公共节点（例如Infura）通常不支持你直接使用它的节点管理账户，因此必须 自己离线创建账户。一个简单的例子就是Metamask钱包。Metamask就是使用的 Infura公共节点，因此它采用的是离线方式创建账户；与之相对应，如果你部署 私有节点，那么即可以使用节点软件创建账户，当然也可以离线创建账户。

以太坊的[eth\_accounts](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/3/7/) 接口用来获取节点管理的账户，在web3j中，对应的封装方法为ethAccounts， 响应对象为EthAccounts：



例如，下面的代码获取节点管理的账户列表：

Request**<?**,EthAccounts> request = web3j.ethAccounts();

EthAccounts ethAccounts = request.send();

**List**<String> accounts = ethAccounts.getAccounts();

或者采用链式写法简便地改写为：

List<String> accounts = web3j.ethAccounts().send().getAccounts();

参考教程，为App.java实现以下功能：

* 创建web3j对象
* 在控制台显示当前所连接节点旳全部账号

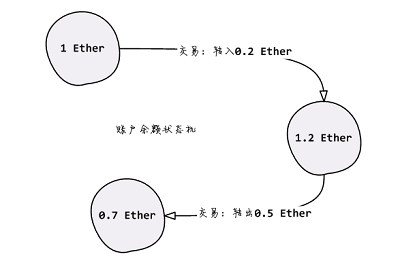
# 状态与交易

## 概述

在这一部分，我们将学习如何使用web3j向以太坊提交交易，具体包括如下内容：

* 了解以太坊状态机的概念
* 理解普通交易与裸交易的区别
* 提交普通交易
* 提交裸交易
* 交易管理器与受控交易

可以把以太坊视为一个分布式状态机，交易则是以太坊发生状态迁移的 原因，也就是说，交易总是以修改状态为目的。以太坊中包含成千上万 的状态，账户余额就是其中最著名的一个：

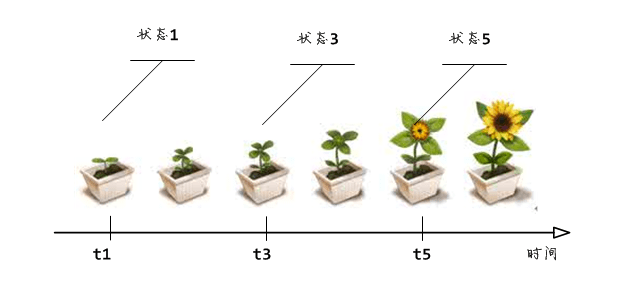


在以太坊中，外部应用可以向节点提交两种交易：普通交易和裸交易。 普通交易由节点签名，而裸交易则由外部应用负责签名。相应的，这两种 交易需要调用不同的接口进行提交。

由于共识机制，向节点提交的交易不会立刻生效，最理想的情况下是被 矿工置入下一个新块。因此交易提交后还需要耐心等待交易凭证。为了简化 两步交易代码，web3j引入了交易管理器与受控交易的概念。

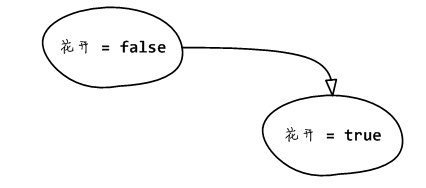
## 状态与激励

如果你把时间想象成离散的，就容易理解什么是状态：

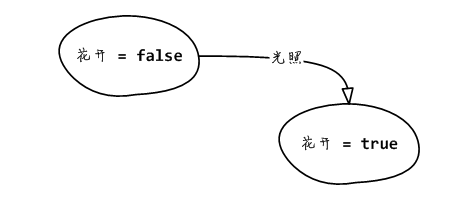


简单的说，状态就是某一个时刻你对所关注的世界的抽象表达的确定值， 而这个值，随时间的延伸或外部的影响可能发生变化。

不同的人眼中有不同的世界，因此也会有不同的抽象表达。例如，如果 我们关注花有没有开，那么我们对这个世界的抽象表达使用一个布尔变量 就够了：在t1 ~ t4，状态的值都是false，而t5和t6时，状态的值则是true， 剔除时间的因素，可以用下面的状态迁移图来压缩表示：



显然，我们关注的世界从一个状态迁移到另一个状态，一定有其原因， 例如，充分的光照。导致状态迁移的原因，在状态迁移图中标注为激励：



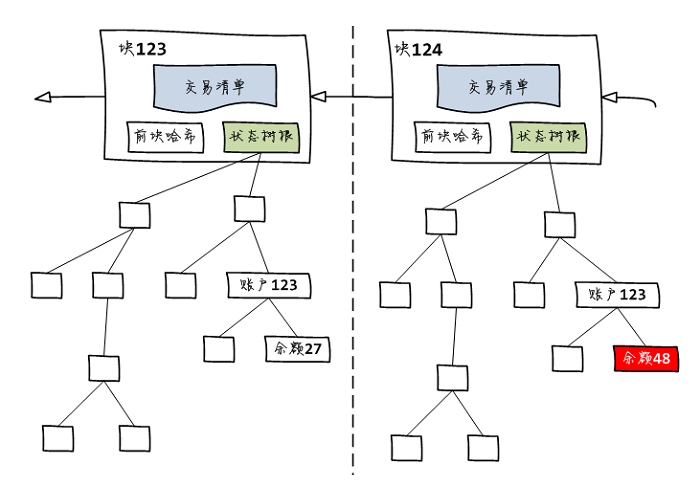
有限数量的状态与激励引起的状态迁移，这种模型在计算机领域被称为 有限状态机（FSM），而有限状态机正是以太坊的核心计算模型。

阅读教程回答问题：假如你关注的是花的大小，请画出相应的状态迁移图。

## 以太坊状态机

以太坊将世界抽象为数量巨大的状态节点，而交易则是导致这些状态节点 的值发生变化的激励。

例如，某个账户的余额就是一个状态节点，它的值在某个时刻是确定的， 如果其他人向该账户发起转账交易，那么在下一个时刻，这个状态节点 旳值就会更新为新的值：



以太坊采用一种树形数据结构来保存所有的状态节点，该状态树的根节点则和交易记录一起保存在区块中。在每一个区块都对应着一棵状态树，它代表着世界在那个时刻的确定状态 —— 一个快照。

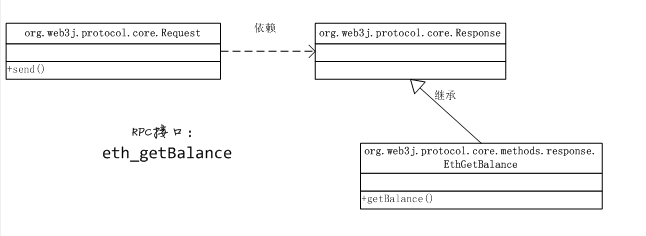
容易理解，状态可以从交易推演出来。例如，要得到一个账户的余额，只需要把该账户所有的转账交易汇总在一起就可以得出。但以太坊通过使用状态树来表征交易的结果，使得这一查询可以迅速完成。

阅读教程回答问题：

* 在以太坊区块中存有状态的具体值吗？
* 可以检索到一个账户的余额的历史变化情况吗？

## 获取账户余额

以太坊定义了[eth\_getBalance](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/3/8/) 接口用来获取账户余额，在web3j中，对应的封装方法为ethGetBalance()， 响应对象为EthGetBalance：



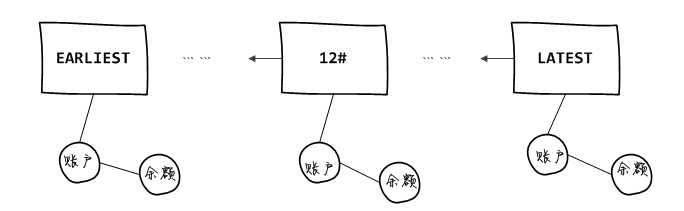
例如，下面的代码读取节点管理的第一个账户的余额：

String account = web3j.ethAccounts().send().getAccounts().get(0);

BigInteger balance = web3j.ethGetBalance(account,

DefaultBlockParameterName.LATEST).send().getBalance();

注意ethGetBalance()方法的第二个参数，使用它来指定一个特定的块。DefaultBlockParameterName.LATEST这个值表示使用链上的最后一个块，也就 意味着它使用最后一个块的状态树中记录的账户余额，即当前账户余额：



由于每个块都有对应着块生成那一时刻的状态树，因此你指定一个不同的块，就意味着将账户余额这个状态回溯到那个块生成的特定时刻。例如，查看这个账户 最初的余额：

String account = web3j.ethAccounts().send().getAccounts().get(0);

BigInteger balance = web3j.ethGetBalance(account,

DefaultBlockParameterName.EARLIEST).send().getBalance();

也可以使用编号来指定块，例如，查看第12块时的账户余额：

String account = web3j.ethAccounts().send().getAccounts().get(0);

BigInteger balance = web3j.ethGetBalance(account,

new DefaultBlockParameterNumber(12L)).send().getBalance();

参考教程，在App.java中实现以下功能：

* 显示节点管理的第6个账户的当前余额
* 显示该账户在初始时刻的余额

## 以太坊货币单位

如果你成功打印输出了节点账户的余额，可能会吓一跳，怎么这么多钱！

balance@latest: 100000000000000000000

嗯，通货膨胀 :|)

ethGetBalance()方法返回的数字，其单位为wei，是以太币众多面值单位中 最小的一只，而我们常说的1个以太币，则对应单位ether，这两者之间差着 18个0：

1 ether = 1000000000000000000 wei

因此上面的账户余额，按ether计算，就是100个以太币了 —— 默认情况下，ganache 创建的测试账户自动充值100个以太。

以太坊定义了10个等级的面值单位，连续两个单位之间总是差3个0，也就是1000倍 的关系。下表列出了常用的单位：

| **单位** | **单位价值** | **换算为wei** |
| --- | --- | --- |
| wei | 1 wei | 1 |
| Kwei(babbage) | 1e3 wei | 1,000 |
| Mwei(lovelace) | 1e6 wei | 1,000,000 |
| Gwei(shannon) | 1e9 wei | 1,000,000,000 |
| microether(szabo) | 1e12 wei | 1,000,000,000,000 |
| milliether(finney) | 1e15 wei | 1,000,000,000,000,000 |
| ether | 1e18 wei | 1,000,000,000,000,000,000 |

值得指出的是，这些货币单位的名称，大多取自在计算机与密码学领域做出杰出 贡献的科学家的名字。例如：wei取自Wei Dai，戴伟，密码学家，B-Money发明人：



这个有必要顶一下，华人的骄傲！

其他的还有：

* lovelace：Ada Lovelace，诗人拜伦之女，据称是第一个女程序员
* babbage：Charles Babbage，英国数学家、发明家、机械工程师，提出了 差分机与分析机的概念设计，被视为计算机先驱
* shannon：Claude Elwood Shannon，美国数学家、电子工程师和密码学家， 现代信息论的创始人
* szabo：Nick Szabo，密码学家，智能合约概念的提出者
* finney：Hal Finney，密码学家，提出了工作量证明机制（POW）

ether则取自从古希腊开始就被认为是构成天体的神秘物质，一直到19世纪科学家都认为宇宙中充满了以太，认为光是通过以太传播的 —— 当然，现在这种观点已经被 抛弃了。

阅读教程回答问题：

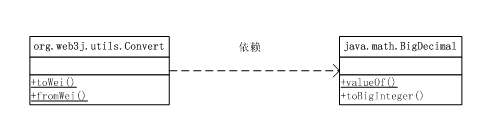
* 单位 wei取自哪位科学家的名字？
* 1 Gwei 可以换算成多少个 wei？

## 货币单位表示与换算

在org.web3j.utils.Convert类中，定义了枚举类型Unit来表示以太坊 的各种货币单位，例如：

* Convert.Unit.WEI
* Convert.Unit.KWEI
* Convert.Unit.MWEI
* Convert.Unit.GWEI
* Convert.Unit.SZABO
* Convert.Unit.FINNEY
* Convert.Unit.ETHER
* Convert.Unit.KETHER
* Convert.Unit.METHER
* Convert.Unit.GETHER

由于在运算中通常使用wei作为计量单位，Convert类同时也提供了两个静态 方法用于将其他单位换算到wei，或者反之：



例如，下面的代码将1个ether为wei计量的值：

BigDecimal oneEther = Convert.toWei("1",Convert.Unit.ETHER);

下面的代码将1 wei转换为ether计量的值：

BigDecimal oneWei = Convert.fromWei("1",Convert.Unit.ETHER);

如果你需要进行转换的两个单位中不包含wei，那么可以结合以上 两个函数，先转化到wei，再转化到目标单位。

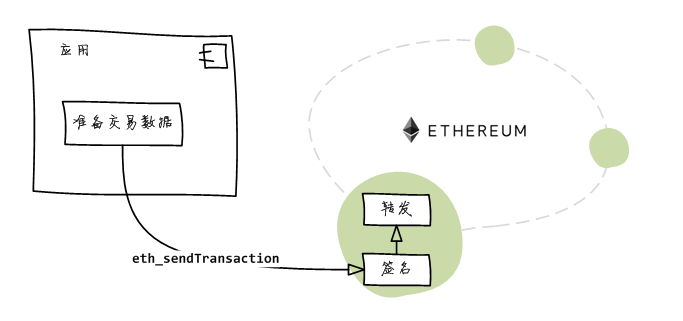
参考教程，在App.java中实现如下功能：

* 将20GWei转换为Wei
* 将5Finney转换为GWei

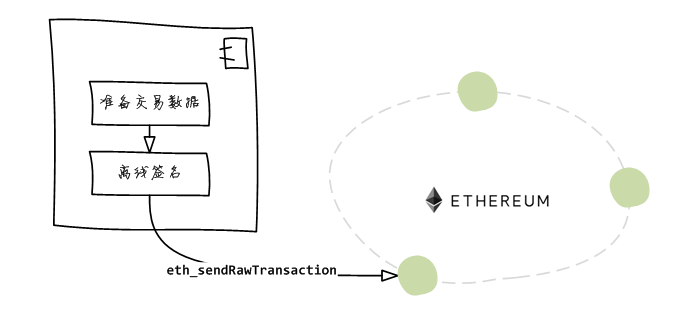
## 交易类型

在以太坊中，约定了两种交易：普通交易（Transaction）和裸交易（RawTransaction）。

这两种交易的区别在于：普通交易由节点负责签名，然后发送到网络中进行确认；



而裸交易则由外部应用进行签名，节点不再额外处理，而只是负责发送到网络中进行确认 —— 这也是裸交易名称的由来 —— 未经节点加工的原始交易：



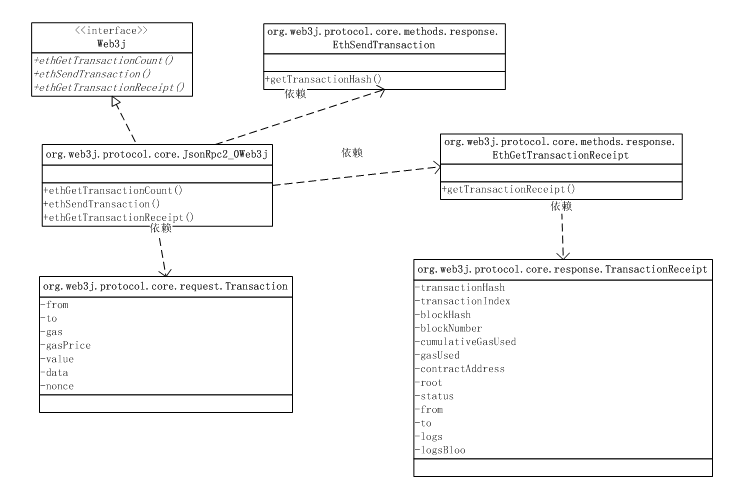
以太坊约定了两种交易不同的提交接口：普通交易使用eth\_sendTransaction 调用提交，而裸交易则应当使用eth\_sendRawTransaction调用提交。事实上， 在公共节点中，通常会拒绝普通交易的提交，而要求外部应用必须进行离线签名。

阅读教程回答问题：

* 普通交易和裸交易的区别在哪里？
* 如果连接到自己部署的节点，可以使用普通交易吗？

## 提交普通交易

普通交易由节点负责进行签名。在web3j中，提交一个普通交易，需要使用ethSendTransaction()方法发送一个Transaction对象，该方法对应[eth\_sendTransaction](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/3/17/) 这个RPC接口：



应用首先应当准备一个Transaction对象，该对象中包含了四种信息：

* 交易基本信息：发送方、接收方、发送金额
* 有偿执行交易的报酬信息：gas价格、gas用量上限
* 交易标识信息：nonce，用来对抗重放攻击
* 补充数据信息：交易额外携带的数据，也将写入区块永存

不过只有发送方和接收方的信息是必须的，其他不需要的信息都可以设置为null。例如， 下面的代码从节点第1个账户向第2个账户转100wei的资金：

List<String> accounts = web3j.ethAccounts().send().getAccounts();

String **from** = accounts.get(0);

String to = accounts.get(1);

BigInteger gasPrice = null;

BigInteger gasLimit = null;

BigInteger value = BigInteger.valueOf(100L); *//wei*

BigInteger nonce = null;

String data = null;

Transaction tx = **new** Transaction(**from**,nonce,gasPrice,gasLimit,to,value,data);

String txHash = web3j.ethSendTransaction(tx).send().getTransactionHash();

向节点成功发送交易对象后，节点将返回该交易的哈希值。我们将在 下一节课程中，使用这个交易哈希值，来检查交易的执行状态。

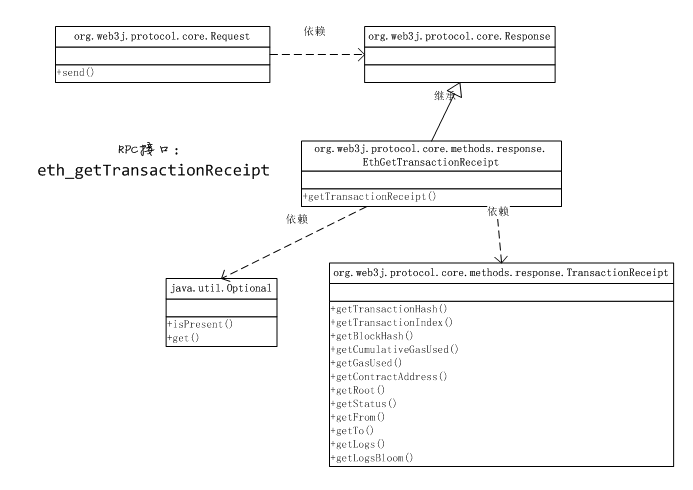
参考教程，在App.java中实现以下功能：

* 从节点第3个账户向第5个账户转1个ether
* 在控制台显示交易哈希

## 获取交易收据

由于以太坊的交易需要提交到网络中进行共识处理，因此我们提交的交易不会 马上生效。要查询交易是否生效，需要使用ethSendTransaction()方法返回 的交易哈希读取交易收据。

根据以太坊的约定，应用需要调用[eth\_getTransactionReceipt](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/3/26/) 接口来检索具有指定哈希交易的收据。在web3j，该请求对应ethGetTransactionReceipt() 方法，响应则对应EthGetTransactionReceipt类：



例如，下面的代码尝试读取具有指定哈希的交易的收据：

Optional<TransactionReceipt> receipt = web3j.ethGetTransactionReceipt(txHash)

.send().getTransactionReceipt();

按照以太坊的出块速度，大约最快需要15秒交易才可能得到确认，因此我们需要 周期性地检查交易收据。例如，下面的代码每隔10秒钟检查一次交易收据， 直到超时或取得有效收据：

**long** timeout = 300000L;

**long** t0 = System.currentTimeMillis();

Optional<TransactionReceipt> receipt = **null**;

**while**(**true**){

receipt = web3j.ethGetTransactionReceipt(txHash).send().getTransactionReceipt();

**if**(receipt.isPresent()) { **break**;}

**if**((System.currentTimeMillis() - t0) > timeout) { **break**; }

Thread.sleep(1000\*10);

}

当然，在使用ganache时，由于出块速度非常块，可以提高检查频率，例如每隔1秒 检查一次即可。

参考教程，在App.java中实现以下功能：

* 从节点第3个账户向第5个账户转1个ether
* 在控制台显示交易哈希
* 轮询该交易的收据
* 在控制台显示交易收据内容

## gas价格与用量

在我们之前创建交易对象时，有意忽略了gas相关的参数，让节点自己决定。 因为，gas是以太坊中最令人迷惑的概念之一。

**Gas**：Gas对应于一个交易(Transaction)中以太坊虚拟机(EVM)的实际运算步数。 越简单的交易，例如单纯的以太币转帐交易，需要的运算步数越少， Gas亦会需要 的少一点。 反之，如果要计算一些复杂运算，Gas的消耗 量就会大。

**Gas Price**：Gas Price就是你愿意为一个单位的Gas出多少Eth，一般用Gwei作单位。 所以Gas Price 越高， 就表示交易中每运算一步，会支付更多的Eth。

因此，以太坊的交易手续费计算公式很简单：

交易手续费(Tx Fee) = 实际运行步数(Actual Gas Used) \* 单步价格(Gas Price)

例如你的交易需要以太坊执行50步完成运算，假设你设定的Gas Price是2 Gwei ，那么整个 交易的手续费 就是50 \* 2 = 100 Gwei 了。

**Gas Limit**：Gas Limit就是一次交易中Gas的可用上限，也就是你的交易中最多会执行多少步运算。 由于交易复杂程度各有不同， 确切的Gas消耗量是在完成交易后才会知道，因此在你提交交易 之前，需要为交易设定一个Gas用量的上限。

## web3j中为交易设置gas参数

在web3j中创建Transaction对象时，可以根据具体情况设定gas Price为20GWe，设定gas Limit 为60000：

List<String> accounts = web3j.ethAccounts().send().getAccounts();

String **from** = accounts.get(0);

String to = accounts.get(1);

BigInteger gasLimit = BigInterger.valueOf(60000);

BigInteger gasPrice = Convert.toWei("20",Convert.Unit.GWEI).toBigInteger();

BigInteger value = BigInteger.valueOf(100L); *//wei*

BigInteger nonce = null;

String data = null;

Transaction tx = **new** Transaction(**from**,nonce,gasPrice,gasLimit,to,value,data);

String txHash = web3j.ethSendTransaction(tx).send().getTransactionHash();

如果说你提交的交易尚未完成，消耗的Gas就已经超过你设定的Gas Limit，那么这次交易 就会被取消，而已经消耗的手续费同样被扣取 —— 因为要奖励已经付出劳动的矿工。 而如果交易已经完成，消耗的Gas未达到Gas Limit， 那么只会按实际消耗的Gas 收取交易服务费。 换句话说，一个交易可能被收取的最高服务费就是Gas Limit \* Gas​​ Price 了。

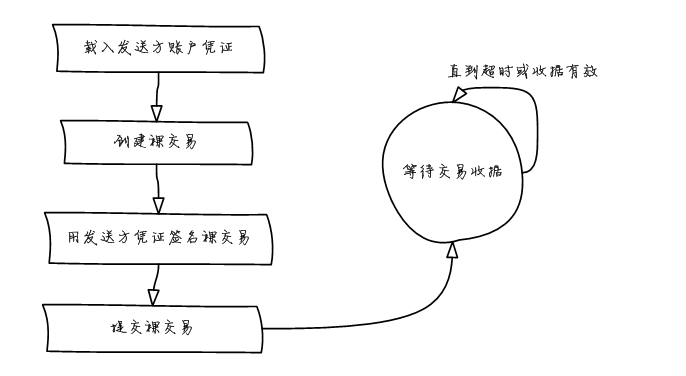
最后值得一提的是Gas Price 越高，你提交的交易会越快被矿工接纳。 但通常人们都不愿多支付手续费， 那么究竟应该将Gas Price设置为多少，才可以在正常时间(eg 10 mins)内，确保交易被确认到区域链上 呢？ [这个网站](http://ethgasstation.info/)可以帮到你。

阅读教程，在以下陈述中选择正确的说法：

* 一个交易实际消耗的gas量与交易的复杂程度有关
* gas price和gas limit的乘积就是你的交易实际要支付的手续费
* 只要gas price给的足够高，你的交易就会优先执行

## 使用裸交易

与普通交易由节点负责签名不同，裸交易需要外部应用进行离线签名。 因此在使用裸交易之前，需要首先载入账户凭证 —— 要用到凭证里 保存的私钥进行签名：



可以使用钱包来实例化一个账户凭证对象，例如，下面的代码使用 密码123解密指定的钱包文件：

Credentials credentials = WalletUtils.loadCredentials("123","./keystore/...");

接下来创建裸交易对象。由于发送方将对裸交易签名，因此在裸交易对象中不需要 重复指定发送方账户。例如，下面的代码将构造一个从钱包账户向节点第2个账户转账 的裸交易对象：

List<String> accounts = web3j.ethAccounts().send().getAccounts();

String to = accounts.get(1);

BigInteger gasPrice = BigInteger.valueOf(22000000000L);

BigInteger gasLimit = BigInteger.valueOf(670000);

BigInteger value = Convert.toWei("1",Convert.Unit.ETHER).toBigInteger();

String data = "";

BigInteger nonce = getNonce(credentials.getAddress());

RawTransaction rawTx = RawTransaction.createTransaction(nonce,gasPrice,gasLimit,to,value,data);

nonce的作用是对抗重放攻击，在不同的交易中它应当是不重复的，除非你需要覆盖之前 的交易。在以太坊中应当将这个值设置为发送方账户已经发送的交易数量 —— 使用 [eth\_getTransactionCount](http://cw.hubwiz.com/card/c/ethereum-json-rpc-api/1/3/10/) 调用来获取这个值：

**function** **getNonce**(String account) **Throws** **Exception** {

**return** web3j.ethGetTransactionCount(account,DefaultBlockParameterName.LATEST)

.send().getTransactionCount();

}

一旦创建了裸交易对象，就可以使用发送方账户对其进行签名，签名的结果是一个字节码流：

byte[] signedMessage = TransactionEncoder.signMessage(rawTx,credentials);

String hexValue = Numeric.toHexString(signedMessage);

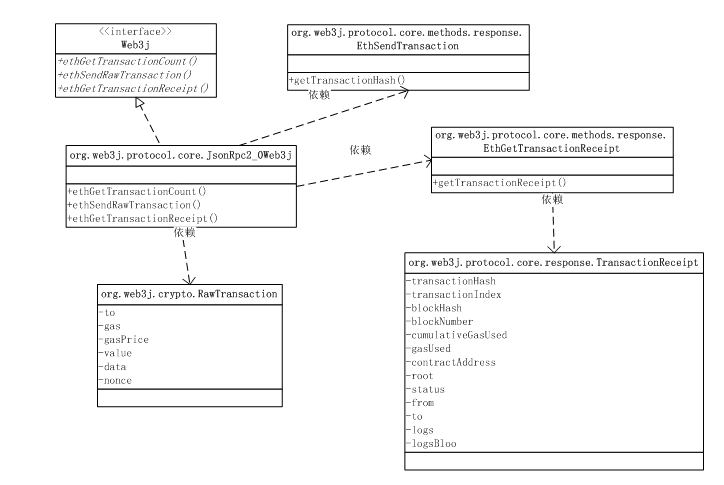
Numeric.toHexSting()方法的作用，是将字节数组转换为带0x前缀的16进制字符串。

一切就绪，使用ethSendRawTransaction()方法来构建裸交易请求对象：

String txHash = web3j.ethSendRawTransaction(hexValue).send().getTransactionHash();

同样，我们也需要等待裸交易生效后的收据。

与裸交易相关的主要类及关系如下图所示：

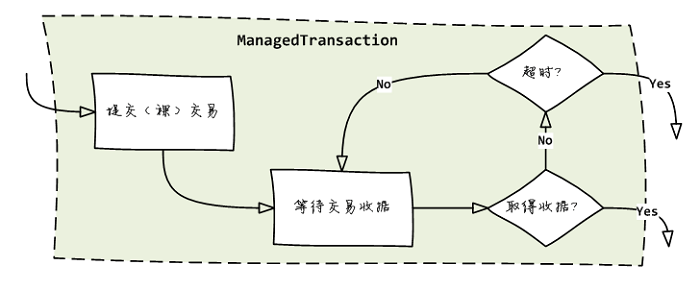


参考教程，在App.java中实现以下功能：

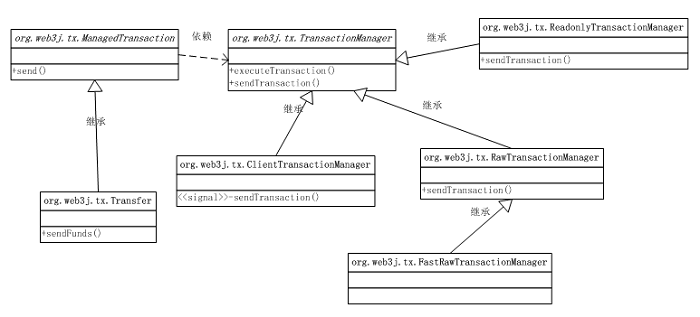
* 创建一个钱包文件
* 从节点第1个账户向该钱包账户转入2个ether
* 从该钱包账户向节点第3个账户转入100wei

## 受控交易与交易管理器

从前面的课程中容易看到，在应用中执行以太坊交易反复出现这样的模式 —— 提交交易、等待收据：



如果把这两个环节封装起来，可以极大地简化应用代码的编写。事实上， 这就是web3j中的受控交易 —— ManagedTransaction。



受控交易是一个抽象类，我们使用Transfer类来展示受控交易类的简洁，它是封装了资金转账逻辑的受控交易类，使用其sendFunds()方法进行资金转账操作。

例如，下面的代码从节点管理的第1个账户向第1个账户转入0.1个ether：

List<String> accounts = web3j.ethAccounts().send().getAccounts();

String **from** = accounts.**get**(0):

String to = accounts.**get**(1);

BigDecimal **value** = BigDecimal.valueOf(0.1);

ClientTransactionManager ctm = **new** ClientTransactionManager(web3j,**from**);

Transfer transfer = **new** Transfer(web3j,ctm);

TransactionReceipt receipt = transfer.sendFunds(to,**value**,Convert.Unit.ETHER).send();

Transfer类实现了受控转账交易，其sendFunds()方法返回一个RemoteCall对象该返回对象的send()方法将真正触发以太坊交易的发送。

可以看到，我们在调用sendFunds().send()提交交易后，直接得到了交易收据，这意味着，这个调用的内部，帮我们进行了繁琐的收据等待处理。

由于我们使用节点管理的账户，因此使用ClientTransactionManager，其构造函数需要传入发送方的账户。

ClientTransactionManager ctm = **new** ClientTransactionManager(web3j,**from**);

对交易收据的等待是在交易管理器里完成的。在交易管理器中将在发送交易后启动一个线程等待并最终返回交易收据。

默认情况下，web3j交易管理器将以15秒的周期最多检查40次交易收据。15秒是以太坊默认的出块时间。你可以在创建交易管理器时修改这两个参数。例如，下面的代码将以1秒的周期检查10次 —— 在使用仿真器时，这是合理的参数：

**int** **from** = accounts.**get**(0):

**int** attempts = 10;

**int** sleepDuration = 1000; *//ms*

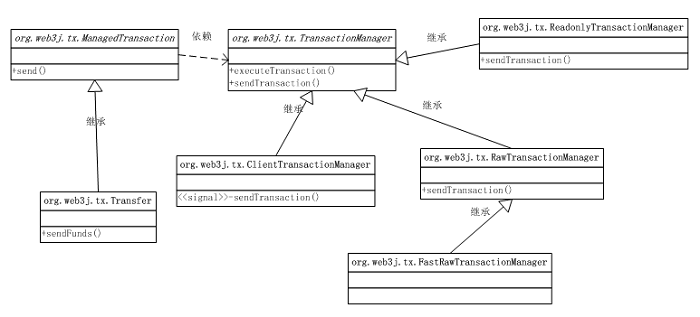
ClientTransactionManager ctm = **new** ClientTransactionManager(web3j,**from**,attempts,sleepDuration);

参考教程，在App.java实现以下功能：

* 使用Transfer类从节点第3个账户向第4个账户转入0.2个ether
* 调整上述受控交易的收据等待行为，以1秒周期最多检查10次收据

## 裸交易管理器

web3j的交易管理器不仅封装了两步交易操作，而且还使用不同的继承类封装了 不同的交易签名方式：



例如，在上一节使用的ClientTransactionManager类，它将向节点提交普通 交易，由节点负责签名后再发送到网络中。因此，在它的构造函数中，只需要 声明执行账户即可：

String account = accounts.**get**(0);

ClientTransactionManager ctm = **new** ClientTransactionManager(web3j,accounts.**get**(0));

与此对应，RawTransactionManager类则使用应用管理的私钥执行离线签名， 然后提交裸交易给节点。因此，在它的构造函数中，需要指定的是账户凭证 Credentials对象：

String pass = "123";

String walletPath = "./keystore/...";

Credentials credentials = WalletUtils.loadCredentials(pass,walletPath);

RawTransactionManager rtm = new RawTransactionManager(web3j,credentials);

同样，你也可以调节裸交易管理器的收据等待行为：

**int** attempts = 10;

**int** sleepDuration = 1000; *//ms*

RawTransactionManager rtm = **new** RawTransactionManager(web3j,credentials,attempts,sleepDuration);

参考教程，在App.java中使用受控交易实现以下功能：

* 创建一个钱包文件
* 从节点第8个账户向该钱包账户转入2个ether
* 从该钱包账户向节点第2个账户转入20个Gwei

# 智能合约

## 概述

智能合约就是区块链上运行的软件，它常常被类比为「自动贩卖机」，因为大家认为这样比较容易理解： 自动贩卖机能接受并执行外部的指令。当顾客选定商品并付款后， 自动贩卖机将释放商品给顾客， 并不需要额外的人工介入：



智能合约的概念最早由电脑科学家、密码学家Nick Szabo在1994年提出， 不过当时并没有合适的环境实现。由于区块链上的交易具有可追溯、抗篡改、不可逆转的特性， 使智能合约在没有第三方中间人的情况下，也可以进行安全的交易，这才使得自动化执行的智能合约得以落地。

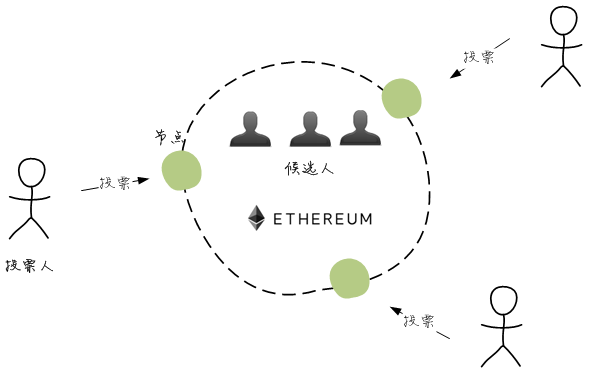
而以太坊由于内置了虚拟机和开发语言，这使得在以太坊区块链上开发智能合约的效率大大提高、难度大大降低。因此，现在提到智能合约，基本上大家说的都是以太坊上的智能合约。

在这一部分的课程中，我们将学习以下内容：

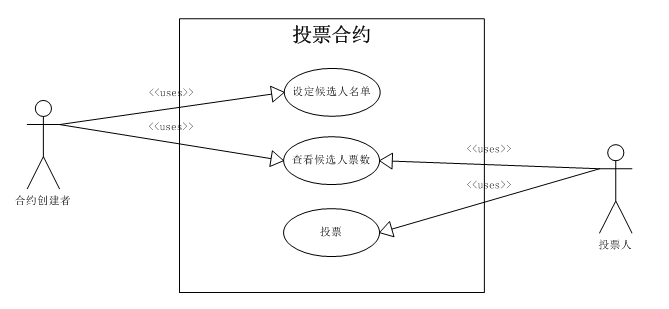
* 使用solidity开发一个投票智能合约
* 使用命令行工具将solidity智能合约编译、转化为java包装类
* 编写部署合约的java代码
* 在java代码中与智能合约交互

## 去中心化投票

投票是集体决策的一种重要形式，在去中心化的系统中，由于剔除了中心化 的权威机构，来自不同个体的投票在很多应用中都发挥着重要的作用。下图是基于以太坊的去中心化投票的示意图：



在这个去中心化投票系统中有两个角色：合约创建者和投票人。候选人名单是固定 的，由合约创建者在部署合约时设定，我们使用字符串来表示候选人；投票人则只能投票给名单中的候选人，我们使用以太坊账户来表示投票人；合约创建者和投票人都可以查看候选人的得票数：

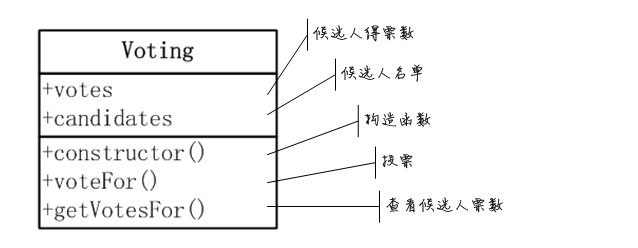


阅读教程，回答以下问题：

* 为什么在用例分析中没有将候选人设定为一种角色？

## 设计合约接口

Solidity合约非常类似于Java中的类：属性用来声明合约的状态，方法用来修改或 读取状态。因此我们很容易根据前一节的用例设计出投票合约的主要接口：



我们肯定需要记录每个候选人的得票数，这就是votes状态的作用；而 candidates状态则用来记录所有的候选人名称。

voteFor()方法是为投票人服务的，而getVoteFor()方法则用来获取某个 候选人的得票数，至于设定候选人名单，我们将在构造函数中完成。

阅读教程，回答以下问题：

* 如果使用java实现Voting类，你会使用什么数据结构来实现votes属性 和candidates属性？
* 基于你的数据结构选择，如何实现voteFor()和getVoteFor()方法？

## 编写合约代码

首先我们创建合约文件Voting.sol。.sol后缀表示这是一个solidity代码文件。 遵循gradle的目录约定，我们将这个文件放在src/main/resources/solidity目录下：

~/repo/hello$ mkdir -p src/main/resources/solidity

~/repo/hello$ touch src/main/resources/solidity/Voting.sol

实现代码及注释如下：

pragma solidity >=0.4.24; *//编译器版本要求：0.4.x*

contract Voting { *// 使用contract关键字定义合约*

mapping (bytes32 => uint8) **public** votes; *//使用映射表记录候选人得票数*

bytes32[] **public** candidates; *//使用定长数组记录候选人名单*

*//构造函数。参数：候选人名单*

constructor(bytes32[] memory candidateNames) **public** {

candidates = candidateNames;

}

*//获取指定候选人得票数。参数：候选人名称。返回值：8位无符号整数*

**function** **getVotesFor**(bytes32 candidate) **view** **public** **returns** (uint8) {

**require**(validCandidate(candidate));*//要求指定名称必须是有效候选人，否则停止执行*

**return** votes[candidate];

}

*//投票给指定候选人。参数：候选人名称*

**function** **voteFor**(bytes32 candidate) **public** {

**require**(validCandidate(candidate));

votes[candidate] += 1;

}

*//检查指定的名称是否在候选人名单里。参数：候选人名称。返回值：true或false*

**function** **validCandidate**(bytes32 candidate) **view** **public** **returns** (bool) {

**for**(uint i = 0; i < candidates.length; i++) {

**if** (candidates[i] == candidate) {

**return** **true**;

}

}

**return** **false**;

}

}

合约文件的第一句总是用来声明对编译器的要求，solidity中可以使用语义化版本号。 例如：^0.4.24将匹配所有0.4.x版本的编译器。

bytes32[]表示定长字符串（32字节）数组，在Solidity中由于bytes32比string类型 有更小的计算开销，因此我们使用定长字符串来表示候选人名称。

mapping (bytes32 => uint8)定义了一个从定长字符串（键）到无符号8位整数（值） 的映射表类型，因此它很合适记录候选人的得票数。Solidity的映射表非常类似于java中的 哈希表，但不同的是，你无法枚举映射表的键。这是因为映射表实际上是记录的键哈希 到值的映射，因此它只能进行单向检索。

Solidity使用constructor来声明合约的构造函数，这是我们设置候选人名单的时机。

getVotes()方法用来读取指定候选人的得票数，由于这个函数不会修改合约的状态， 因此在函数声明时，使用了view修饰符：

**function** **getVotesFor**(bytes32 candidate) **view** **public** **returns** (uint8){}

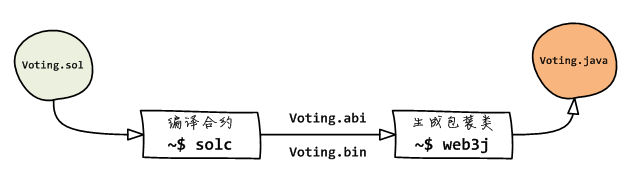
该函数同时在声明时使用returns关键字定义了返回值。

在voteFor()和getVoteFor()函数实现内，都使用require()调用来检查 输入条件是否满足函数执行的要求。require语句类似于java中的断言assert， 如果条件不满足，则不会继续执行，但也不会消耗调用账户的gas。

参考教程和示例代码，编写投票合约代码。

## 生成合约包装类

为了在Java代码中与合约交互，我们需要将solidity合约转化为Java类，流程如下：



## 编译合约

solidity编译器可以从合约源代码编译出EVM字节码和二进制应用接口（ABI）。 字节码是最终运行在以太坊虚拟机中的代码，而ABI则是描述合约接口的一个JSON 对象，用来在其他开发语言中调用合约。

例如，下面的命令使用--bin和--abi选项声明同时生成字节码和ABI对象， 并输出到build目录：

solc:

~/repo/hello$ solc src/main/resources/solidity/Voting.sol --bin --abi --optimize --overwrite -o src/main/resources/solidity/build/

编译成功后，将在build目录中得到两个文件：

~/repo/hello$ ls src/main/resources/solidity/build

Voting.abi Voting.bin

接下来，我们将使用这两个文件来生成Java中可以使用的包装类。

|  |
| --- |
| 编译过程修改为如下方式：  1、安装solidity编译工具，npm install -g solc  2、cd进入到Voting.sol所在目录；  3、solcjs –-bin Voting.sol 生产EVM字节码（注意是两个杠）  4、solcjs -–abi Voting.sol 生产应用接口ABI（注意是两个杠）  这样在文件夹中就对应生成bin、abi两个文件。 |

## 生成包装类

一旦生成了合约的字节码和ABI，就可以使用web3j提供的命令行工具来生成包装类。

例如，下面的代码使用之前生成的字节码文件和abi文件，在com.hubwiz.demo.contracts 包中生成对应的包装类：

~/repo/hello$ web3j solidity generate src/main/resources/solidity/build/Voting.bin src/main/resources/solidity/build/Voting.abi -p com.hubwiz.demo.contracts -o src/main/java/

现在，就可以在代码中使用Voting类了。例如，在你的java代码中：

**import** **com**.hubwiz.demo.contracts.Voting;

参考教程，为投票合约生成java包装类。

如何安装web3j命令行：

1、去<https://github.com/web3j/web3j/releases/tag/4.3.0> 下载[web3j-4.3.0.zip](https://github.com/web3j/web3j/releases/download/4.3.0/web3j-4.3.0.zip)包后解压，在bin目录中可以看到web3j.bak文件。

2、生产包装类的命令如下：

|  |
| --- |
| web3j solidity generate  -b F:\git-code\zuk-p2p\zuk-p2p\zuk-p2p-client\src\main\resources\solidity\Voting\_sol\_Voting.bin -a F:\git-code\zuk-p2p\zuk-p2p\zuk-p2p-client\src\main\resources\solidity\Voting\_sol\_Voting.abi -p com.huizhiwang.contracts  -o F:\git-code\zuk-p2p\zuk-p2p\zuk-p2p-client\src\main\java |

3、这样就会生产一个对应java文件。

## 编写合约处理脚本

我们可以编写一个脚本，来简化编译合约、生成包装类这一流程：

~/repo/hello$ touch buildContracts.sh

buildContracts.sh内容如下：

**#!/usr/bin/env bash**

set -e

set -o pipefail

*# 合约代码根目录*

srcDir=src/main/resources/solidity

*# 合约编译输出目录*

abiDir=$srcDir/build/

*# java代码根目录*

javaDir=src/main/java

*# 逐个处理合约代码文件*

**for** file **in** `ls $srcDir/\*.sol`; **do**

*# 剔除代码文件名后缀*

target=$(basename $file .sol)

*# 编译合约代码*

echo "Compiling Solidity file ${target}.sol"

solc --bin --abi --optimize --overwrite \

--allow-paths "$(pwd)" \

$file -o $abiDir

echo "Complete"

*# 生成java包装类*

echo "Generating contract bindings"

web3j solidity generate \

$abiDir/$target.bin \

$abiDir/$target.abi \

-p com.hubwiz.demo.contracts \

-o $javaDir > /dev/null

echo "Complete"

**done**

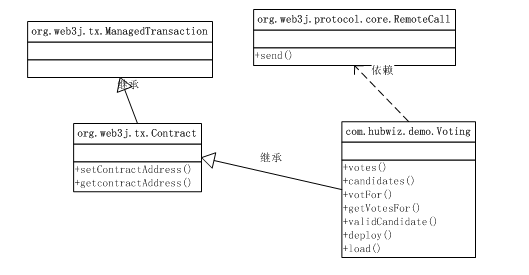
现在，只需执行这个脚本就可以直接编译合约并生成包装类了：

~/repo/hello$ ./buildContracts.sh

参考教程，编写合约处理脚本，并使用该脚本重新处理投票合约。

## 部署/载入合约

生成的合约包装类Voting继承自Contract，而Contract也是一个受控交易对象：



查看生成的Voting.java文件，容易看到与合约中对应的方法，例如voteFor()等，而 合约中的状态则转化为同名函数，例如为votes状态生成的votes()方法。

除此之外，合约包装类中还生成了用于合约部署的静态方法deploy()。例如， 下面的代码部署合约并使用getContractAddress()方法获得合约地址：

*//使用节点账户部署合约*

List<String> accounts = web3j.ethAccounts().send().getAccounts();

ClientTransactionManager ctm = **new** ClientTransactionManager(web3j,accounts.**get**(0));

*//准备参数：候选人名单*

List<**byte**[]> candidates = **new** ArrayList<**byte**[]>();

candidates.**add**(stringToByte32("Tommy"));

candidates.**add**(stringToByte32("Jerry"));

candidates.**add**(stringToByte32("Micky"));

*//部署合约*

Voting voting = Voting.deploy(web3j,ctm,Contract.GAS\_PRICE,Contract.GAS\_LIMIT,candidates).send();

*//获取合约的部署地址*

String contractAddress = voting.getContractAddress();

由于我们在solidity合约中使用bytes32定长字符串，因此需要把Java中的String转化为 定长byte[]数组：

**private** **static** **byte**[] **stringToBytes32**(String str){

**byte**[] a = **new** **byte**[32];

System.arraycopy(str.getBytes(),0,a,32-str.length(),str.length());

**return** a;

}

### 载入已部署合约

我们并不需要每次启动程序都部署一次合约，事实上，更常见的场景， 是在程序启动时载入一个已经部署在链上的合约。

使用合约包装类的load()静态方法载入一个已知地址的已部署合约。例如：

String contractAddress = "0x...";

Voting voting = Voting.load(contractAddress,web3j,ctm,Contract.GAS\_LIMIT);

参考教程，在App.java中实现以下功能：

* 使用节点第1个账户部署投票合约
* 在控制台输出合约的部署地址
* 使用节点第2个账户载入刚才部署的投票合约

## 调用合约方法

web3j在合约包装类中自动生成与solidity合约中对应的方法，我们只需要 直接使用即可。

例如，可以下面的java代码为Tommy投票：

TransactionReceipt receipt = voting.voteFor(stringToBytes32("Tommy")).send();

由于voteFor()方法修改合约的状态，因此该调用提交的是一个交易， 对应于eth\_sendTransaction或eth\_sendRawTransaction调用，封装类的对应 方法返回的总是交易收据对象。

另一种情况，下面的代码读取链上Tommy获得的总票数：

BigInteger votes = voting.getVotesFor(stringToBytes("Tommy")).send();

由于合约的getVotesFor()方法仅读取合约的状态而不进行修改，因此该调用 提交的是消息调用，对应于eth\_call调用，也就是说该方法直接在节点虚拟机中运行而 不必同步到网络中的其他节点。因此，封装类的对应方法可以返回solidity合约中 定义的返回值。

### 使用不同的账户

由于web3j的合约包装类在部署或载入时就绑定了账户，因此如果需要使用多个账户 执行合约方法，例如分别以不同的身份投票，就需要创建不同的包装类实例。

例如，下面的代码分别使用第1个和第2个节点账户进行投票：

String contractAddress = "0x...";

List<String> accounts = web3j.ethAccounts().send().getAccounts();

ClientTransactionManager ctm1 = **new** ClientTransactionManager(web3j,accoutns.get(0));

Voting v1 = Voting.load(contractAddress,web3j,ctm1,Contract.GAS\_PRICE,Contract.GAS\_LIMIT);

v1.voteFor(string2Byte32("Tommy")).send();

ClientTransactionManager ctm2 = **new** ClientTransactionManager(web3j,accoutns.get(1));

Voting v2 = Voting.load(contractAddress,web3j,ctm2,Contract.GAS\_PRICE,Contract.GAS\_LIMIT);

v2.voteFor(string2Byte32("Jerry")).send();

参考教程与示例代码，在App.java中实现以下功能：

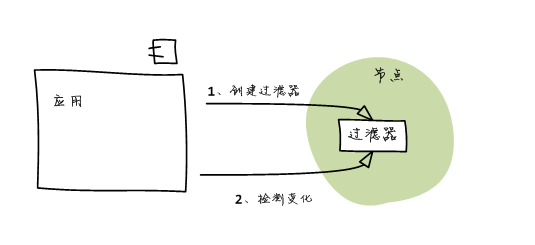
* 使用节点第一个账户为Tommy投票
* 使用节点第二个账户为Jerry投票
* 使用之前创建的钱包账户查看Tommy的得票数

# 过滤器与事件

## 通知机制概述

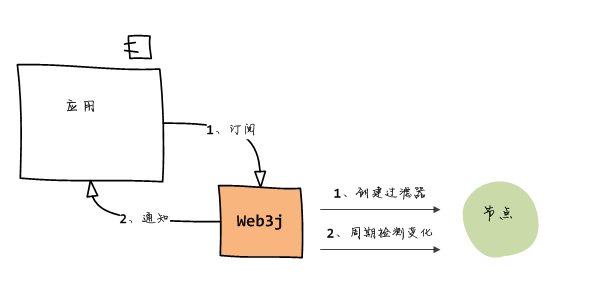
通知机制对任何应用开发都很重要，因为它提供了另外一个方向的变化 通知能力。以太坊也不例外，它的通知机制增强了智能合约与外部应用之间 的沟通能力。

以太坊的通知机制是建立在日志基础之上。例如，如果智能合约触发了一个 事件，那么该事件将写入以太坊日志；如果外部应用订阅了这个事件，那么 应用就可以在日志中出现该事件后及时检测到：



需要指出的是，以太坊的通知机制不是推（Push）模式，而是需要外部应用 周期性轮询的拉（Pull）模式。外部应用通过在节点中创建过滤器来订阅感兴趣 的日志，之后则通过检测该过滤器的变化获得最新的日志。

更经典的通知机制应该是推模式的，因此web3j在此基础上进行了封装，使用RxJava 实现了观察者模式的通知机制：当发生感兴趣的事件时，web3j会自动将事件推送 给应用：



在这一部分的课程中，我们将学习以下内容：

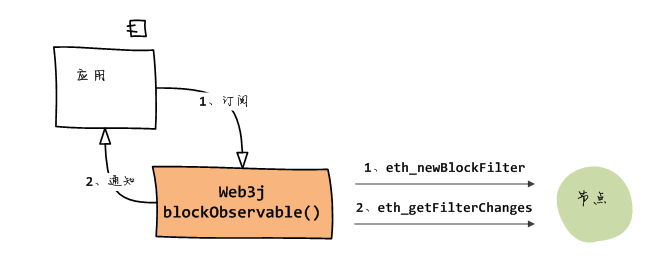
* 使用块过滤器监听新块生成事件和新交易事件
* 使用待定交易过滤器监听待定交易事件
* 使用合约包装类监听合约事件

阅读教程选择以下陈述中正确的说法：

* 以太坊可以主动向外部应用推送事件
* web3j可以主动向java应用推送事件

## 监听新块事件

使用块过滤器来监听新块生成事件。其过程如下：



Web3j的blockObservable()方法将创建一个块事件可观测对象，其参数为true时， 表示在通知事件中将包含完整的交易数据，否则仅包含交易哈希。

例如，下面的代码将在出现新块时打印新块的哈希和编号：

Subscription **sub** = **web3j**.**blockObservable**(**true**).**subscribe**(**ethBlock** -> {

EthBlock.Block block = ethBlock.getBlock();

System.out.println("hash: " + block.getHash());

System.out.println("number: " + block.getNumber());

});

注意通知的参数类型为EthBlock，需要使用getBlock()方法来提取 具体的块信息对象，它的类型是EthBlock.Block。

## 设置轮询周期

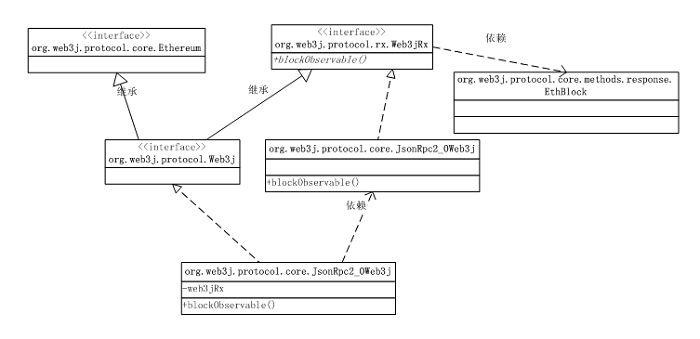
默认情况下，web3j的轮询周期为15秒 —— 以太坊的平均出块时间。如果 我们使用ganache仿真器，会感到事件的触发太慢了 —— 实际上web3j压根 还没有去检查有没有变化的日志。

我们可以在创建Web3j实例时调整这个轮询周期。例如，下面的代码创建 的Webj实例，将每秒检查一次日志变化，这会让事件的触发更加及时：

import org.web3j.utils.Async;

Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"),1000,Async.defaultExecutorService());

web3j的相关类及关系如下图：

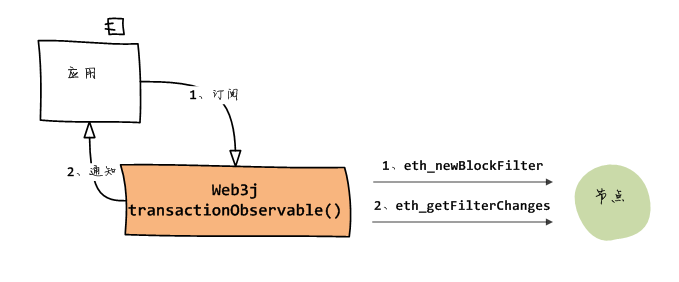


参考教程和示例代码，在App.java中实现以下功能：

* 监听新块生成事件，并打印新块信息
* 从节点第1个账户向第2个账户转入0.1个ether，观察监听的输出

## 监听新交易事件

使用交易过滤器来监听新交易事件。其过程如下：



你可能注意到transactionObservable()的实现也是调用eth\_newBlockFilter 创建的块过滤器，实际上它就是在捕捉到新块事件后，直接展开块中的交易对象。

使用Web3j的transactionObservable()方法创建一个交易事件可观测对象， 其通知参数是Transaction对象。

例如，下面的代码将打印新确认交易的哈希、发送账户和接收账户：

Subscription **sub** = **web3j**.**transactionObservable**().**subscribe**(**tx** -> {

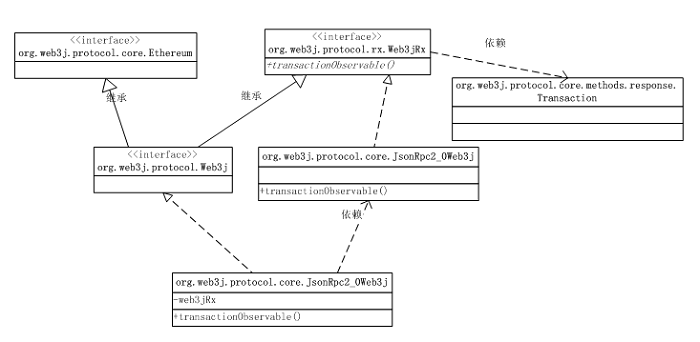
System.out.println("hash: " + tx.getHash());

System.out.println("from: " + tx.getFrom());

System.out.println("to: " + tx.getTo());

});

web3j的相关类及关系如下图：

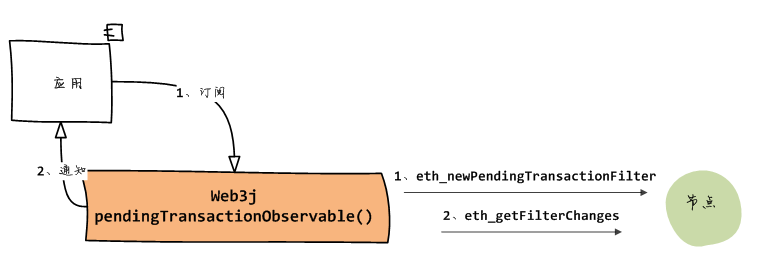


参考教程和示例代码，在App.java中实现以下功能：

* 监听新交易生成事件，显示新交易信息
* 从节点第1个账户向第2个账户转入0.1个ether，查看监听输出

## 监听待定交易事件

使用待定交易过滤器来监听新待定交易事件。其过程如下：



Web3j的pendingTransactionObservable()方法创建一个可观测对象， 它向观察者传入的参数类型为Transaction。

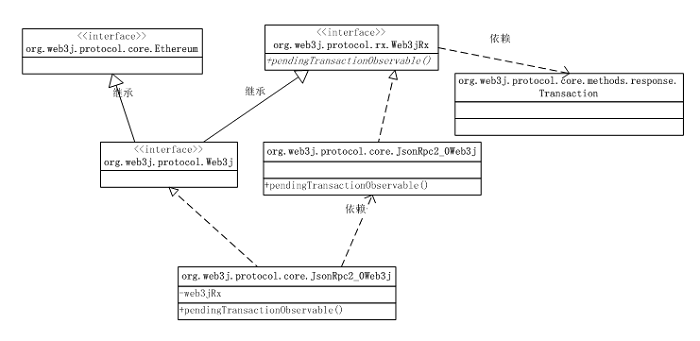
例如，下面的代码将打印新出现的待定交易的哈希：

Subscription **sub** = **web3j**.**pendingTransactionObservable**().**subscribe**(**tx** -> {

System.out.println("hash: " + tx.getHash());

});

web3j的相关类及关系如下：



参考教程和示例代码，在App.java中实现以下功能：

* 监听新待定交易事件，显示待定交易信息
* 从节点第1个账户向第2个账户转入0.1个ether，查看监听输出

## 合约事件

Solidity合约中可以声明事件，并在合适的时机触发事件。

例如，我们可以在投票合约中声明一个Vote事件，当投票时将触发该事件：

pragma solidity ^0.4.24;

contract Voting {

mapping (bytes32 => uint8) **public** votes;

bytes32[] **public** candidates;

*//声明事件Vote*

event Vote(address voter, bytes32 candidate);

constructor(bytes32[] candidateNames) **public** {

candidates = candidateNames;

}

**function** **getVotesFor**(bytes32 candidate) **view** **public** **returns** (uint8) {

**require**(validCandidate(candidate));

**return** votes[candidate];

}

**function** **voteFor**(bytes32 candidate) **public** {

**require**(validCandidate(candidate));

votes[candidate] += 1;

*//触发事件Vote*

emit Vote(msg.sender, candidate);

}

**function** **validCandidate**(bytes32 candidate) **view** **public** **returns** (bool) {

**for**(uint i = 0; i < candidates.length; i++) {

**if** (candidates[i] == candidate) {

**return** **true**;

}

}

**return** **false**;

}

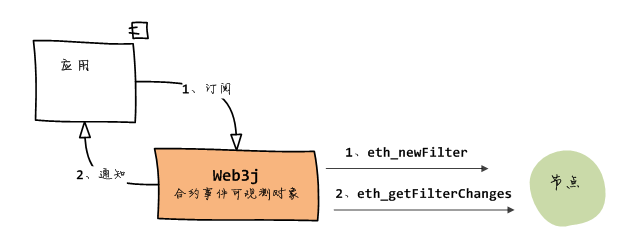
}

参考教程完成以下工作：

* 修改Voting.sol，增加事件声明和触发
* 重新生成java包装类Voting.java
* 部署新合约并在控制台输出部署地址

## 监听合约事件

合约事件的监听是通过主题过滤器实现的，其过程如下：



例如，对于Voting合约中的事件Vote，web3j包装类中将自动为其生成 对应的可观测对象生成函数voteEventObservable()。使用代码如下：

EthFilter filter = **new** EthFilter();

voting.voteEventObservable(filter).subscribe(**event** -> {

System.**out**.println(**event**.voter + " : " + **new** String(**event**.candidate));

});

虽然也可以使用ethLogObservable()方法创建一个主题日志可观测对象 来接收合约事件，例如：

EthFilter filter = **new** EthFilter();

web3j.ethLogObservable(filter).subscribe(log -> {

Systet.out.println("index: " + log.getIndex());

Systet.out.println("data: " + log.getData());

})

但捕捉到的日志还需要进一步解码才可以得到事件的参数，因此，除非特殊情况， 我们总是应该使用包装类中的功能。

参考教程，为App.java实现以下功能：

* 使用节点第1个账户载入之前部署的合约
* 分别使用包装类的方法和主题日志可观测对象监听合约的Vote事件
* 为Micky投票，观察监听的输出，查看两种监听的输出区别