**Android-Debug终端远程调试框架**

2018年4月23日

许启强

**项目介绍**

一般，我们操作手机中的文件是很麻烦的，比如调试数据库，通常来说是如下几种方式：

* 将手机中的SQLite数据库导出到电脑，通过电脑端的软件来查看这个数据库，执行相关的SQL语句，看结果如何。
* Root手机，在手机上安装RE文件管理器，进入应用程序的包下，找到你的数据库的文件，然后再查看数据库中。
* Android Studio有相关的插件，方便操作，但是有的需要收费，使用起来也不是很爽。

现在，利用Android-Debug库，我们可以通过浏览器方便的查看的数据库啦，并且可以执行SQL语句。

Android-Debug允许你以非常简单的方式直接在浏览器中查看并操作文件系统、数据库和SharedPreferences，也可以用命令行操作。

**Android-Debug可以做什么**

Android-Debug可以做什么？

* 访问文件系统，打开、删除、下载设备上的文件，也可以上传文件到设备
* 查看你的应用中所有的数据库，也可以查看文件系统中的数据库
* 对你指定的数据库执行SQL语句；对数据库中的数据进行可视化的编辑；将数据库直接下载下来
* 查看和修改应用程序中使用的共享首选项(SharedPreferences)中的所有数据
* 直接进行shell命令行操作

所有这些功能都可以在不影响设备的情况下工作，不需要root设备（如果有root权限功能将会更加强大）。

本项目在[AMIT SHEKHAR](https://github.com/amitshekhariitbhu)的开源项目Android-Debug-Database基础上进行了重构，增强了运行稳定性，并增加了文件管理系统和命令行系统等。

**使用方式**

**简单使用**

下载[AndroidDebug](https://xuqiqiang.github.io/2017/10/15/debug-2017/#resource_code)项目源码，直接在app–>build.gradle中引入依赖

debugCompile project(path: ':libdebug')

debugCompile的作用：只在你debug编译时起作用，当你release的时候就没必要使用它了。

如果需要在release后也使用到AndroidDebug，就使用compile:

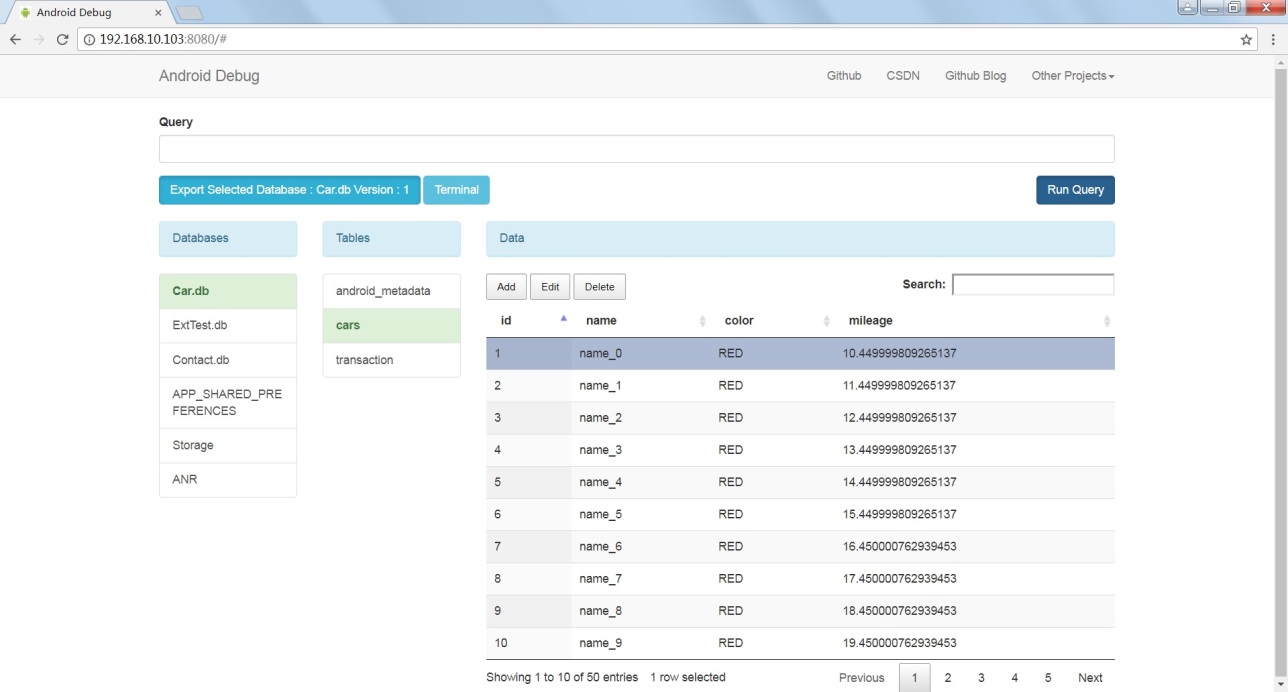
compile project(path: ':libdebug')

这就完了，你不需要任何其他的代码啦。

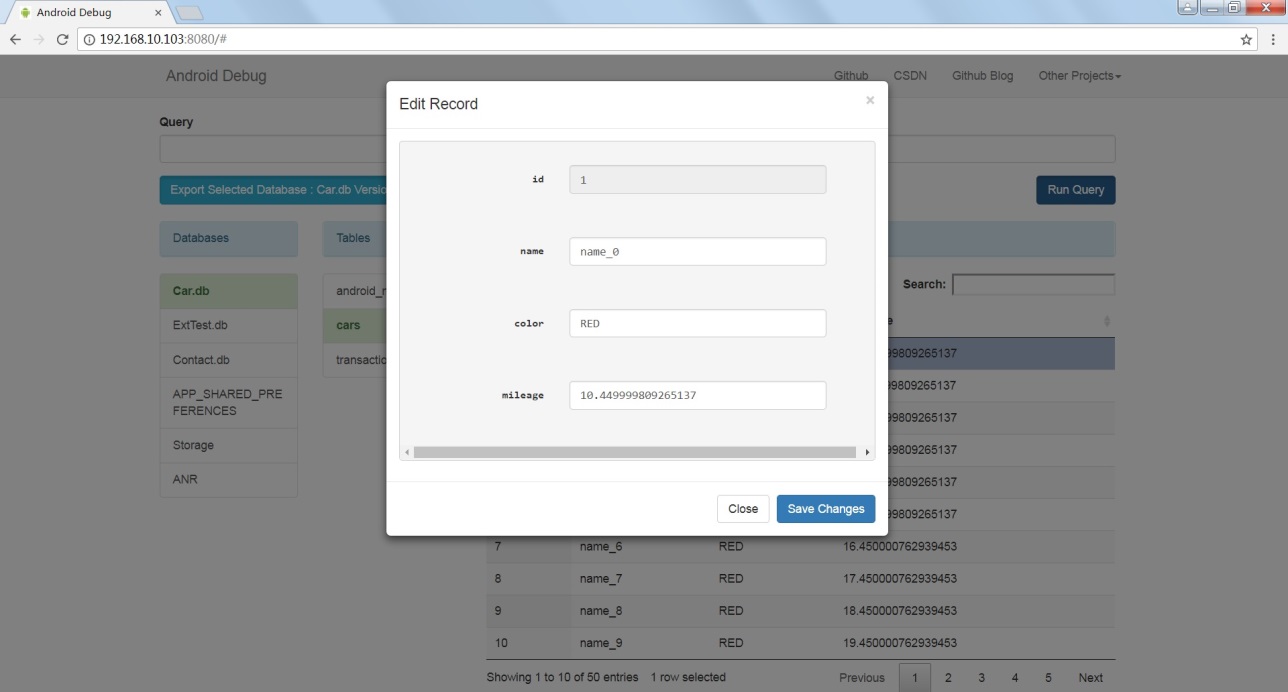
下面当你在App启动的时候，你要注意查看下你的logcat，会有这么一行：

D/libdebug: Open http://XXX.XXX.X.XXX:8080 in your browser

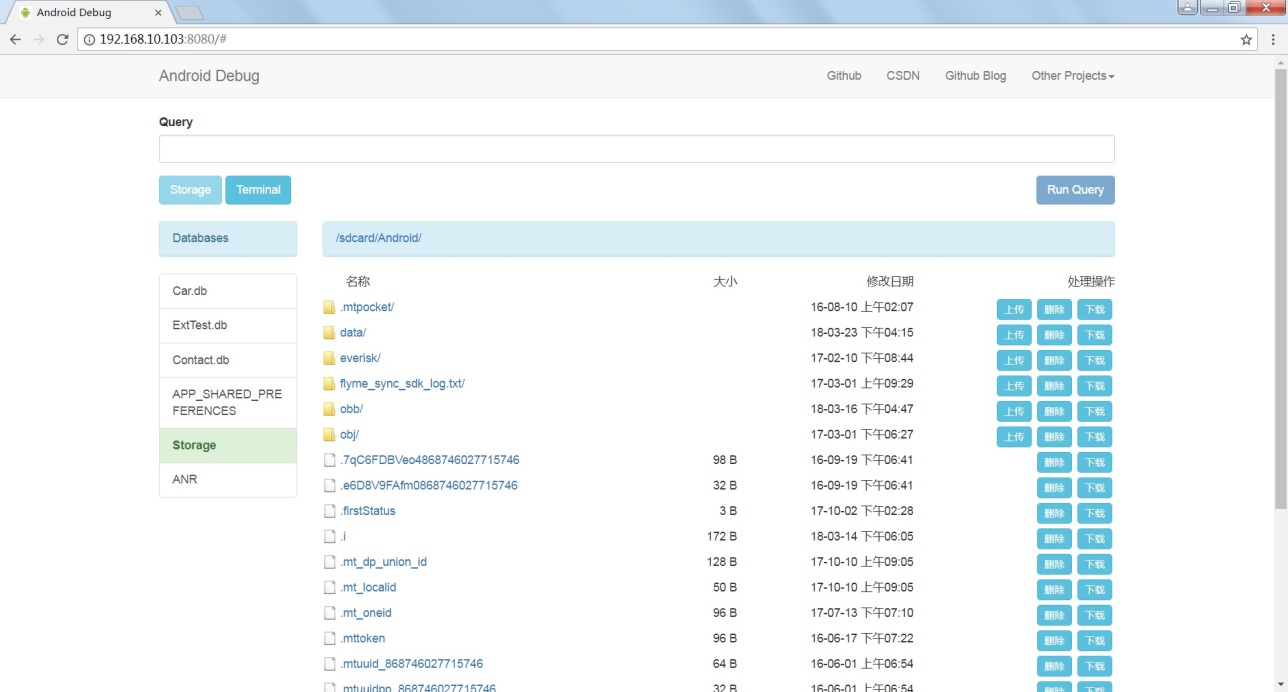
把它复制到你电脑的浏览器，你就可以调试你的设备和应用，注意：你的手机要和电脑在同一个局域网下



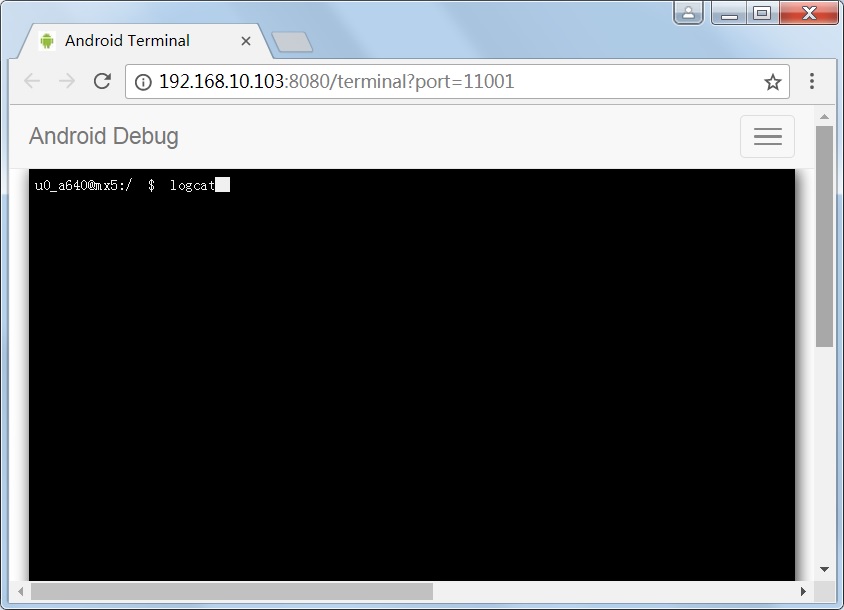
数据库可视化编辑：



文件管理系统：



命令行系统：



**使用远程调试**

**服务端配置**

下载[远程代理服务端](https://xuqiqiang.github.io/2017/10/15/debug-2017/#resource_code)项目源码，编译源码，或者直接使用distribution下面已经生成的程序proxy-server-0.1

将服务端程序拷贝到服务器中

修改conf/config.properties，将server.bind和config.server.bind改为服务器ip，例如服务器ip是192.168.10.102:

server.bind=192.168.10.102

server.port=4900

server.ssl.enable=true

server.ssl.bind=0.0.0.0

server.ssl.port=4993

server.ssl.jksPath=test.jks

server.ssl.keyStorePassword=123456

server.ssl.keyManagerPassword=123456

server.ssl.needsClientAuth=false

config.server.bind=192.168.10.102

config.server.port=8090

config.admin.username=admin

config.admin.password=admin

运行bin/startup.sh（如果是Windows系统则运行bin/startup.bat）

**设备端配置**

回到设备端项目，在app–>build.gradle中，修改buildTypes内部配置:

buildTypes {

debug {

resValue("string", "REMOTE\_PROXY\_INET\_HOST", "119.23.104.192") // 远程代理服务器的ip，将来可能会改成域名的形式

resValue("string", "REMOTE\_PROXY\_INET\_PORT", "4900") // 远程代理服务器的端口

resValue("string", "REMOTE\_PROXY\_WEB\_PORT", "8090") // 远程代理服务器的web网站端口

resValue("string", "DEBUG\_DEV\_MODE", "true") // 打印libdebug日志

resValue("string", "REMOTE\_PROXY\_DEV\_MODE", "true") // 打印libremoteproxy日志

}

release {

resValue("string", "REMOTE\_PROXY\_INET\_HOST", "119.23.104.192")

resValue("string", "REMOTE\_PROXY\_INET\_PORT", "4900")

resValue("string", "REMOTE\_PROXY\_WEB\_PORT", "8090")

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

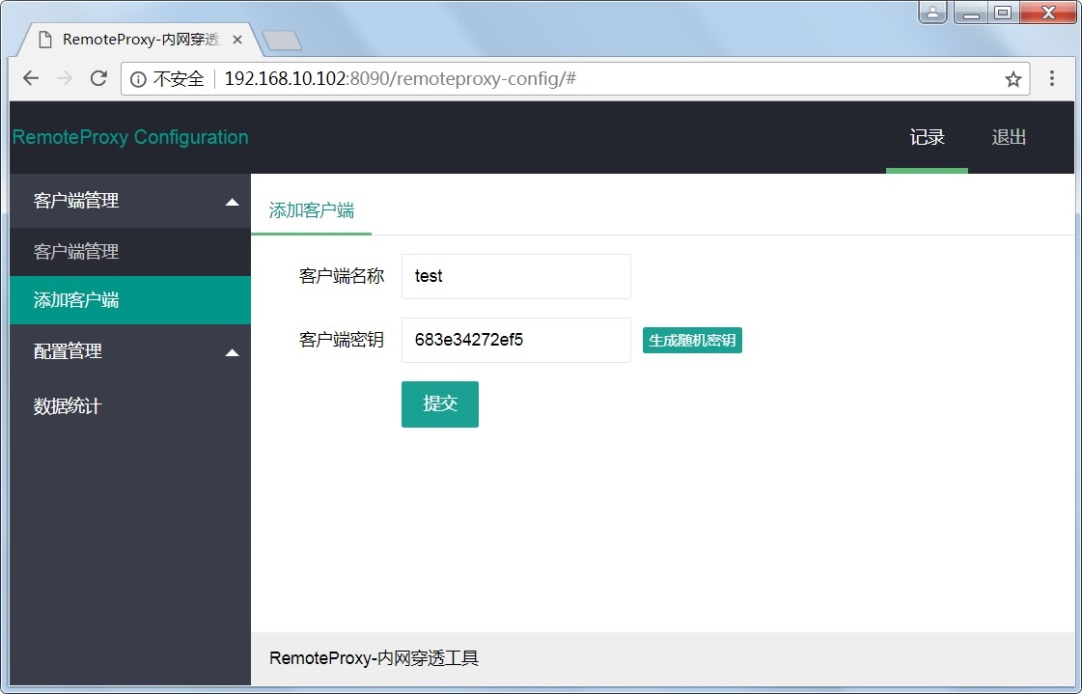
}

设备端运行后，电脑浏览器访问119.23.104.192:8090，效果如图：

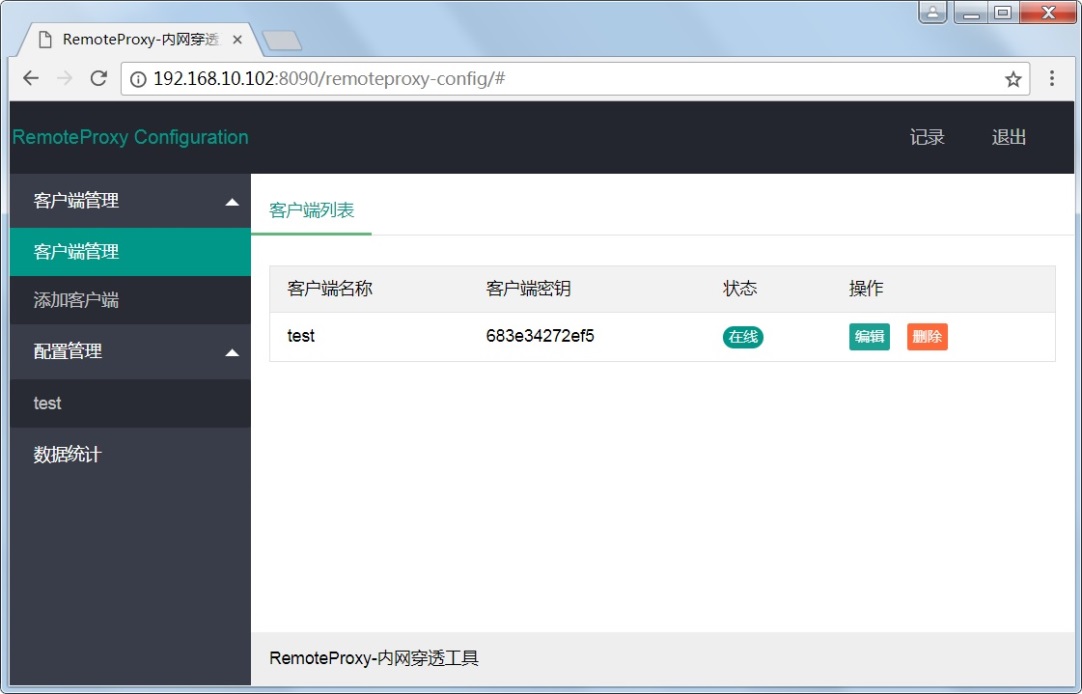
下图中的代理ip有误，不用管。



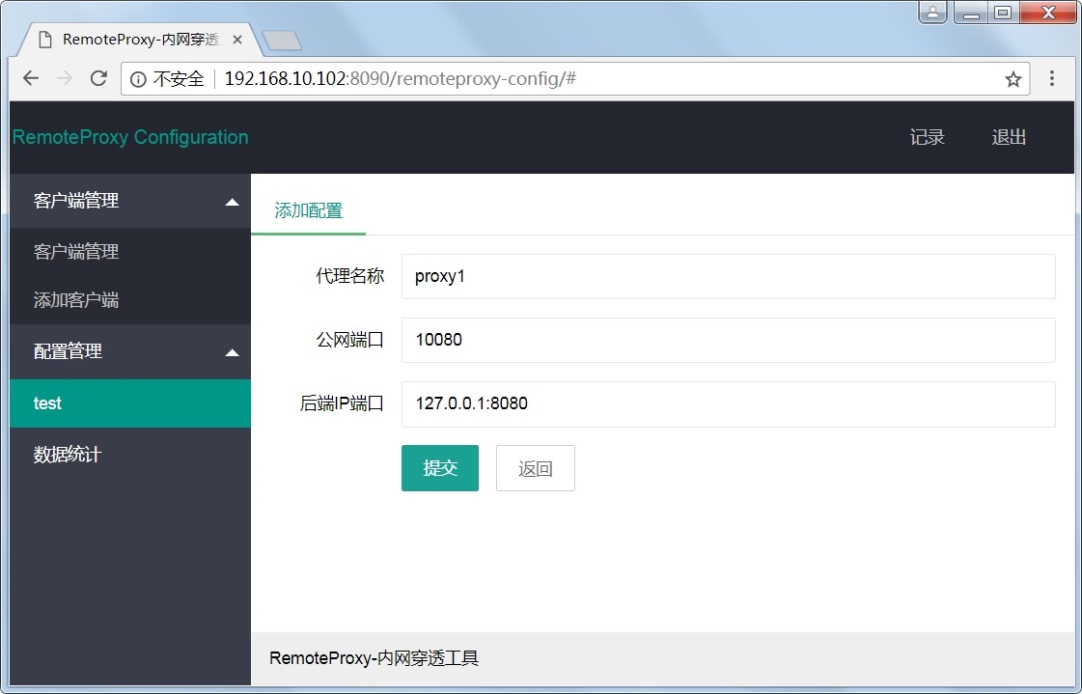
用之前配置的账号进入登录，进入管理系统，点击添加客户端，在客户端密钥填写设备端id，即设备mac：



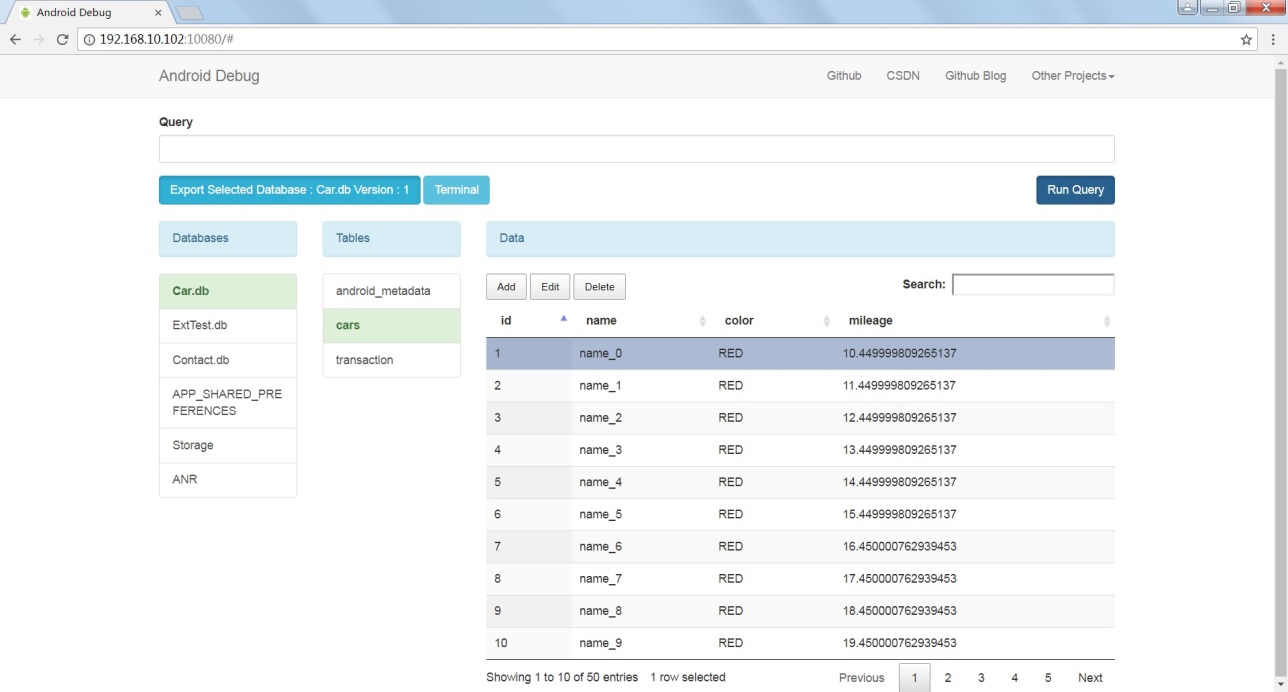
点击提交后，我们将在客户端管理中看到设备信息，不久后状态会自动变为在线：



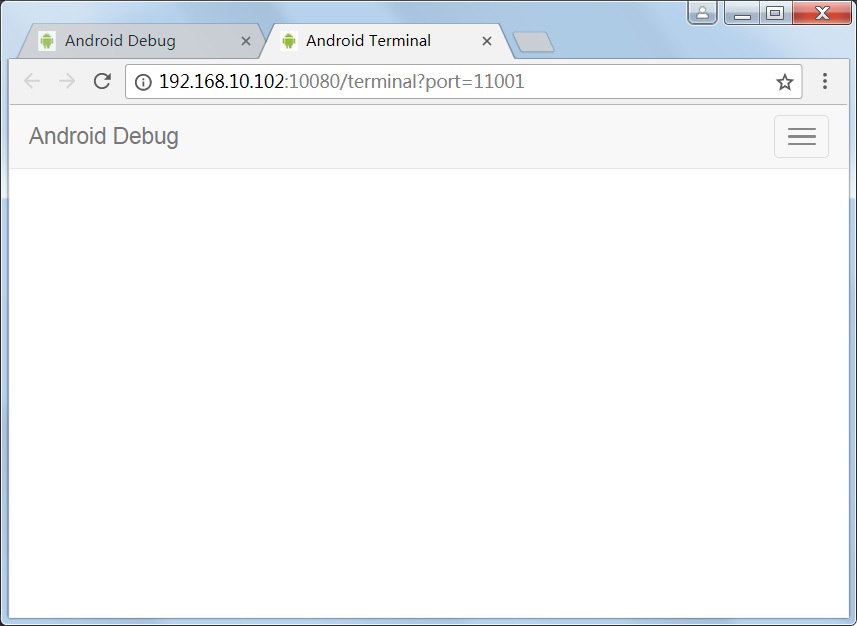
点击配置管理下面的设备名称，点击添加配置，填写如下信息，其中公网端口可以随机设定，比如10080：



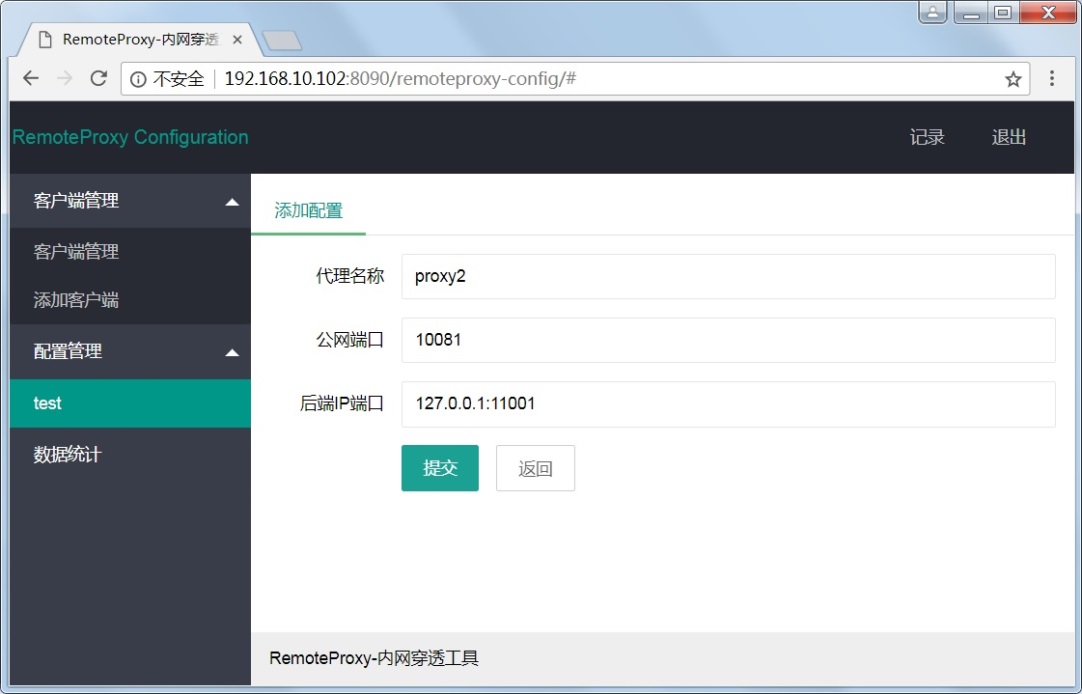
电脑浏览器访问http://192.168.10.102:10080，效果如图：



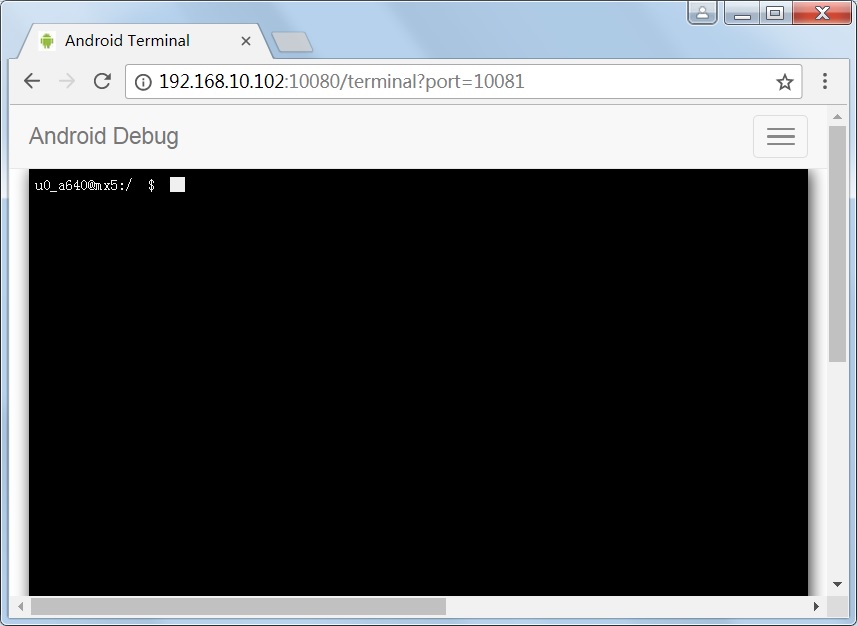
如果需要使用terminal，需要添加新的配置。在网页中点击Terminal，会进入如下页面，记下”port=”后面的数字11001：



回到配置管理，点击添加配置，填写如下信息，其中公网端口也可以随机设定，比如10081：



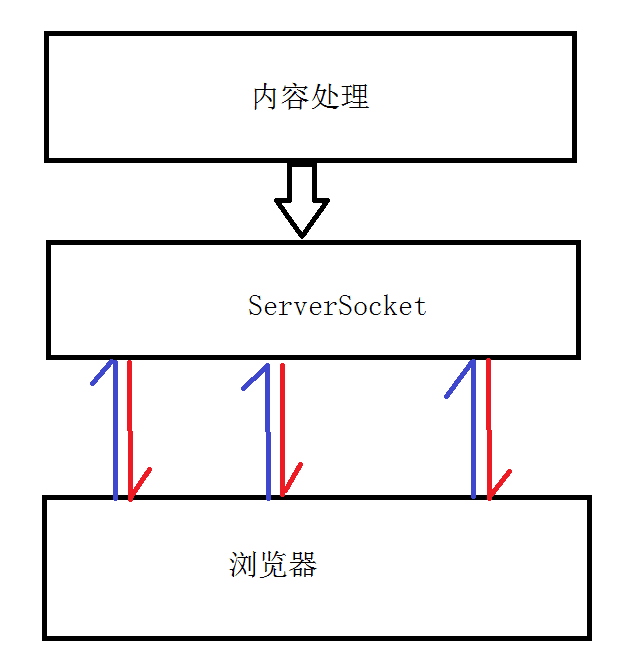
电脑浏览器访问http://192.168.10.102:10080/terminal?port=10081，效果如图：



**原理介绍**

**设备端服务器搭建**

整体结构如下：



项目使用ServerSocket建立了设备服务器端，监听设备上的一个特定端口。当一个远程客户端尝试这个端口时，服务器就会被唤醒，协商建立客户端与服务器端的连接，并返回一个常规的Socket对象，表示2台主机之间的Socket。也是就说服务器端Socket接受到客户端Socket发送过来的连接时，服务器端会生成一个常规的Socket对象，用于向客户端发送数据，数据总是通过常规socket进行传输。

*// 创建服务器套接字*

serverSocket **=** **new** ServerSocket**(**port**);**

*// 设置端口重用*

serverSocket**.**setReuseAddress**(true);**

*// 创建HTTP协议处理器*

BasicHttpProcessor httpproc **=** **new** BasicHttpProcessor**();**

*// 增加HTTP协议拦截器*

httpproc**.**addInterceptor**(new** ResponseDate**());**

httpproc**.**addInterceptor**(new** ResponseServer**());**

httpproc**.**addInterceptor**(new** ResponseContent**());**

httpproc**.**addInterceptor**(new** ResponseConnControl**());**

*// 创建HTTP服务*

HttpService httpService **=** **new** HttpService**(**httpproc**,**

**new** **DefaultConnectionReuseStrategy(),** **new** DefaultHttpResponseFactory**());**

*// 创建HTTP参数*

HttpParams params **=** **new** BasicHttpParams**();**

params**.**setIntParameter**(**CoreConnectionPNames**.**SO\_TIMEOUT**,** 5000**)**

**.**setIntParameter**(**CoreConnectionPNames**.**SOCKET\_BUFFER\_SIZE**,** 8 **\*** 1024**)**

**.**setBooleanParameter**(**CoreConnectionPNames**.**STALE\_CONNECTION\_CHECK**,** **false)**

**.**setBooleanParameter**(**CoreConnectionPNames**.**TCP\_NODELAY**,** **true)**

**.**setParameter**(**CoreProtocolPNames**.**ORIGIN\_SERVER**,** "WebServer/1.1"**);**

*// 设置HTTP参数*

httpService**.**setParams**(**params**);**

*// 创建HTTP请求执行器注册表*

HttpRequestHandlerRegistry reqistry **=** **new** HttpRequestHandlerRegistry**();**

*// 增加HTTP请求执行器*

reqistry**.**register**(**UrlPattern**.**DOWNLOAD**,** **new** HttpDownHandler**(**webRoot**));**

reqistry**.**register**(**UrlPattern**.**DELETE**,** **new** HttpDelHandler**(**webRoot**));**

reqistry**.**register**(**UrlPattern**.**UPLOAD**,** **new** HttpUpHandler**(**webRoot**));**

reqistry**.**register**(**UrlPattern**.**PROGRESS**,** **new** HttpProgressHandler**());**

reqistry**.**register**(**UrlPattern**.**EDIT**,** **new** HttpEditHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**GET\_DB\_LIST**,** **new** HttpGetDbListHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**GET\_DB\_DATA**,** **new** HttpGetDbDataHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**GET\_TABLE\_LIST**,** **new** HttpGetTableListHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**DOWNLOAD\_DB**,** **new** HttpDownloadDbHandler**(**mContext**,** webRoot**));**

reqistry**.**register**(**UrlPattern**.**ADD\_TABLE\_DATA**,** **new** HttpAddTableDataHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**UPDATE\_TABLE\_DATA**,** **new** HttpUpdateTableDataHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**DELETE\_TABLE\_LIST**,** **new** HttpDeleteTableDataHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**QUERY**,** **new** HttpQueryDataHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**GET\_File\_LIST**,** **new** HttpGetFileListHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**OPEN\_TERMINAL**,** **new** HttpOpenTerminalHandler**(**mContext**));**

reqistry**.**register**(**UrlPattern**.**TERMINAL**,** **new** HttpTerminalHandler**(**webRoot**));**

reqistry**.**register**(**UrlPattern**.**BROWSE**,** **new** HttpFBHandler**(**webRoot**));**

*// 设置HTTP请求执行器*

httpService**.**setHandlerResolver**(**reqistry**);**

*// 回调通知服务开始*

**if** **(**mListener **!=** **null)** **{**

mListener**.**onStarted**();**

**}**

*/\* 循环接收各客户端 \*/*

isLoop **=** **true;**

**while** **(**isLoop **&&** **!**Thread**.**interrupted**())** **{**

*// 接收客户端套接字*

Socket socket **=** serverSocket**.**accept**();**

*// 绑定至服务器端HTTP连接*

DefaultHttpServerConnection conn **=** **new** DefaultHttpServerConnection**();**

conn**.**bind**(**socket**,** params**);**

*// 派送至WorkerThread处理请求*

Thread t **=** **new** WorkerThread**(**httpService**,** conn**,** mListener**);**

t**.**setDaemon**(true);** *// 设为守护线程*

pool**.**execute**(**t**);** *// 执行*

**}**

程序根据不同的请求设置了相应的HTTP请求执行器，如HttpGetFileListHandler：

*/\*\* \* 获取文件列表请求处理 \* <p> \* Created by xuqiqiang on 2017/04/17. \*/*

**public** **class** **HttpGetFileListHandler** **implements** HttpRequestHandler **{**

**private** Context mContext**;**

**public** **HttpGetFileListHandler(**Context context**)** **{**

**this.**mContext **=** context**;**

**}**

@Override

**public** **void** **handle(**HttpRequest request**,** HttpResponse response**,** HttpContext context**)**

**throws** HttpException**,** IOException **{**

HttpGetParser parser **=** **new** HttpGetParser**();**

Map**<**String**,** String**>** params **=** parser**.**parse**(**request**);**

String path **=** params**.**get**(**"path"**);**

**if** **(**path **==** **null)** **{**

response**.**setStatusCode**(**HttpStatus**.**SC\_BAD\_REQUEST**);**

**return;**

**}**

path **=** URLDecoder**.**decode**(**path**,** Config**.**ENCODING**);**

String result **=** DatabaseHandler**.**getInstance**(**mContext**).**getFileListResponse**(**path**);**

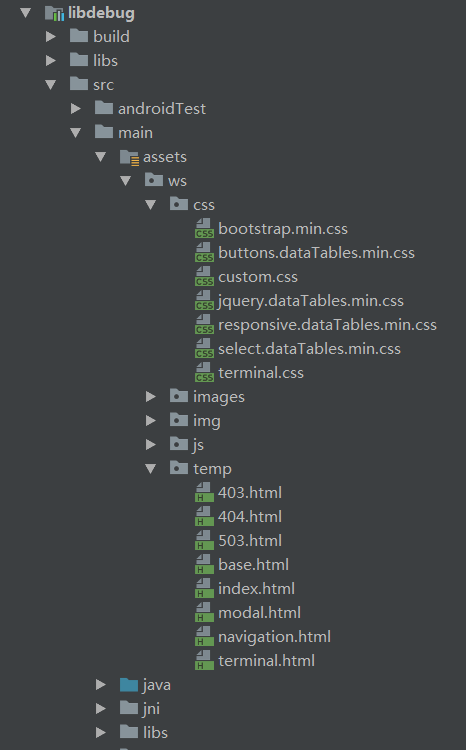
response**.**setEntity**(new** StringEntity**(**result**,** Config**.**ENCODING**));**

**}**

**}**

浏览器中的内容从哪里来的？为啥它能发送请求数据？

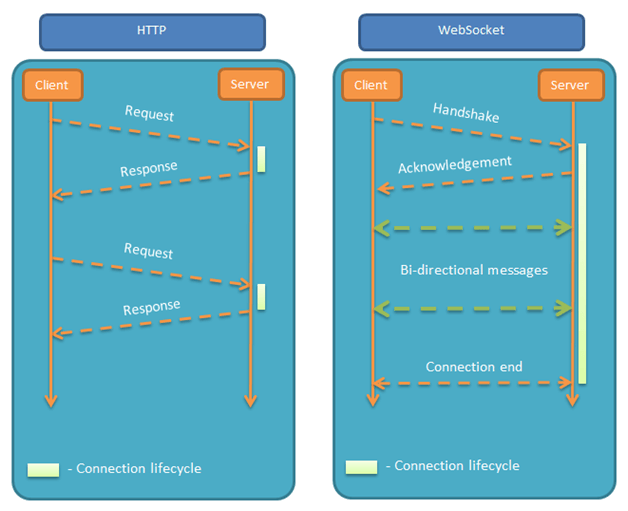
libdebug将具有交互性的html发送给了浏览器：



**命令行系统**

**WebSocket简介**

WebSocket协议是基于TCP的一种新的网络协议。它实现了浏览器与服务器全双工(full-duplex)通信——允许服务器主动发送信息给客户端。



特点包括：

* 建立在 TCP 协议之上，服务器端的实现比较容易。
* 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
* 数据格式比较轻量，性能开销小，通信高效。
* 可以发送文本，也可以发送二进制数据。
* 没有同源限制，客户端可以与任意服务器通信。
* 协议标识符是ws（如果加密，则为wss），服务器网址就是URL。

项目基于c语言实现了WebSocket服务端，运行于设备端；前端基于javascript实现了WebSocket客户端，运行于浏览器网页。

**function** openTerminal(ip, port) {

**var** term;

**var** wsUrl **=** "ws://" **+** ip **+** ":" **+** port

websocket **=** **new** WebSocket(wsUrl);*//new 一个websocket实例*

websocket.onopen **=** **function**(evt) {*//打开连接websocket*

term **=** **new** Terminal(100, parseInt(window.screen.availHeight**/**20), **function**(key) {

websocket.send(key);

console.log("send:" **+** key);

});

term.open();

$('.terminal').detach().appendTo('#container-terminal');

websocket.onmessage **=** **function**(evt) {*//接收到数据*

console.log("receive : " **+** evt.data.charCodeAt(0))

term.write(evt.data);*//把接收的数据写到这个插件的屏幕上*

console.log("receive:" **+** evt.data)

}

websocket.onclose **=** **function**(evt) {*//websocket关闭*

term.write("Session terminated");

term.destroy();*//屏幕关闭*

}

websocket.onerror **=** **function**(evt) {*//额处理*

**if** (**typeof** console.log **==** "function") {

console.log(evt)

}

}

}

}