Android 访问Mysql

1. JDBC
2. 服务器作为中介

# 1 教学内容

1. 网络连接要解决的问题：如何连接；连接得到数据后如何处理（如何解析数据）；多线程问题（连接需要时间）
2. 多线程
3. 连接网络（PC上的Tomcat服务器上的网页）
   1. HttpConnection
   2. OkHttp
4. 解析XML数据
5. 在服务器上读取XML数据，传回Android
6. 小例子：Android读取、修改、删除PC机上的数据
7. 读取json
8. 第三方接口

# **2 多线程（教材P392，第10章）**

传统写法

btnThread.setOnClickListener **{** Thread**{** Log.d("network", "this is thread")  
 **}**.start()  
**}**

Kotlin的写法

Start之类都已经写好，只要写逻辑就行

btnThread.setOnClickListener **{** Thread**{** Log.d("network", "this is thread")  
 **}  
}**

系统崩溃，说明不能直接操作UI（此处的Thread的T应该小写）

Thread**{** txtData.setText("This is thread")  
 Toast.makeText(this, "this is thread",

**}**.start()

消息传递方法：

定义一个Handler

val handler = object: Handler(){  
 override fun handleMessage(msg: Message) {  
 when(msg.what){  
 updateText->{  
 txtData.setText("arg1: ${msg.arg1}--arg2:${msg.arg2}")  
 btnThread.*text* = "hello"  
 }  
 }  
 }  
}

线程中进行传递参数：

Thread**{**

val msg = Message()  
 msg.what = updateText  
 msg.arg1 = 100  
 msg.arg2 = 200  
 handler.sendMessage(msg)  
 **}**.start()

Msg.what: 类型，消息码

Msg.what1和what2，传递参数

对象的传递：

msg.obj = "Hello, this is string"

对象的接收：

val str =msg.obj

传递其它参数：

msg.obj = TrainSearchHistory("杭州", "上海")

接收参数：

val info = msg.obj

使用runOnUiThread(P432)

// 调用runOnUiThread  
btnRunOnUiThread.setOnClickListener **{** Thread**{** runOnUiThread**{** txtData.setText("使用runOnUiThread")  
 **}  
 }**.start()  
**}**

# 3 连接网络

## 3.0 准备工作

* 准备IP，PC机与手机在同一网段，互相可以ping通

家庭局域网：PC：192.168.1.105，手机：192.168.1.104

HWNXT:/ $ ping 192.168.1.105

PING 192.168.1.105 (192.168.1.105) 56(84) bytes of data.

64 bytes from 192.168.1.105: icmp\_seq=1 ttl=64 time=56.4 ms

PC机连接手机热点：

PC端：192.168.43.176

手机端：inet addr:192.168.43.1

虚拟机可以使用地址：10.0.2.2

比如：[http://10.0.2.2:8080/AndroidServer/test.jsp](http://localhost:8080/AndroidServer/test.jsp)

* PC端搭载开发环境
  + MySQL存储数据
  + Tomcat做服务器，传送数据

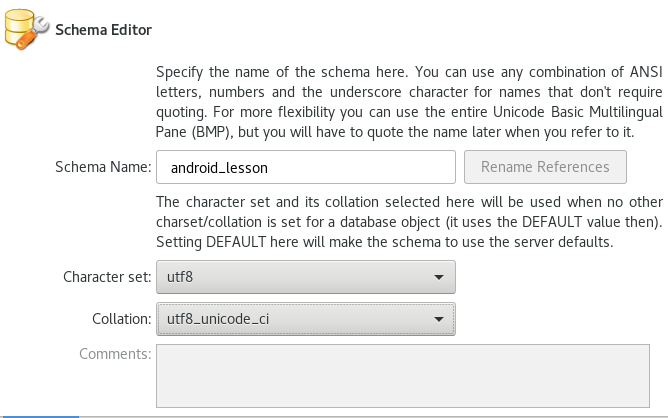
相应的网络：

<http://192.168.1.105:8080/AndroidServer/test.jsp>

* Mysql设置
  + 设置编码

<https://www.cnblogs.com/chyingp/p/mysql-character-set-collation.html>

<https://www.cnblogs.com/EasonJim/p/8128196.html>



Collation: 排序的方式

* + Mysql的JDBC的连接方式已经发生变化

<https://blog.csdn.net/zhangvalue/article/details/85219342>

<https://blog.csdn.net/qq331059279/article/details/99249069>

Mysql8.0的连接方式与Mysql5.0不同

* 打开联网权限

<uses-permission android:name="android.permission.INTERNET"/>

* 开启tomcat的防火墙

<https://www.cnblogs.com/hxlinux/p/12891890.html>

[zafudf@zafudf ~]$ firewall-cmd --add-port=8080/tcp --permanent

success

[zafudf@zafudf ~]$ firewall-cmd --reload

success

## 3.1 HttpUrlConnection HttpClient

// 通过httpURLConn发送请求  
fun sendRequestWithHttpConn(){  
 *thread* **{** var connection : HttpURLConnection ? = null  
 try {  
 val response = StringBuilder()  
 val url = URL(test\_jsp)  
 connection = url.openConnection() as HttpURLConnection  
 connection.*connectTimeout* = 8000 // 连接 超时  
 connection.*readTimeout* = 8000 // read 超时时间  
 val input = connection.*inputStream* // 读取输入流，类似与文件流  
 val reader = BufferedReader(InputStreamReader(input))  
 reader.*use***{** reader.*forEachLine* **{** response.append(**it**)  
 **}  
 }** showResponse(response.toString())  
 }  
 catch (e: Exception){  
 e.printStackTrace()  
 }  
 finally {  
 connection?.disconnect()  
 }  
 **}**}  
  
private fun showResponse(response: String){  
 runOnUiThread**{** tvResponse.*text* = response  
 **}**}

**Tomcat**

**MySql**

**Android**

## 连接网络，请求发送给Web服务器

多线程

线程不允许操作界面

消息机制

1. Web服务器读取数据
2. 回传数据
3. 解析

HttpUrlConnection

**HttpClient**

**OkHttp**

## 3.2 新增书籍

相关的数据库读取、连接已完成

1. 新增add\_book.jsp

不传递参数，在服务端先执行

BookDbBean bean = **new** BookDbBean();

BookInfo info = **new** BookInfo("晋书", "魏征", 1200, 180.5f);

bean.add(info)

1. 在Android端新增调用
2. 添加参数，在服务端执行测试（使用附加参数的方式)

[http://localhost:8080/TestWeb/test/add\_book.jsp?name=后汉书&author=范晔&pages=2000&price=126.8](http://localhost:8080/TestWeb/test/add_book.jsp?name=%E5%90%8E%E6%B1%89%E4%B9%A6&author=%E8%8C%83%E6%99%94&pages=2000&price=126.8)

Android采用地址：

val url = URL("${add\_book}?name=后汉书&author=范晔&pages=2000&price=126.8")

1. 使用output的方式

connection.*requestMethod* = "POST"  
val output = DataOutputStream(connection.*outputStream*)  
output.writeBytes("name=后汉书&author=范晔&pages=2000&price=126.8")

涉及到中文编码问题

问题的解决：

<https://blog.csdn.net/bingjianIT/article/details/51477182>

设置当前编码为UTF-8

connection.setRequestProperty("Accept-Charset", "UTF-8")  
connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded;charset=utf-8")

写入字节数组

val paraStr = "name=后汉书&author=范晔&pages=2000&price=126.8"  
output.write(paraStr.*toByteArray*())

（在java中，中文为一个char，也就是两个字节16位，直接使用writeBytes，会丢弃高8位，只写入低8位的一个字节所以会造成中文乱码。而改正后的方式是先存入字节数组，然后再写入到输出流，不会乱码）

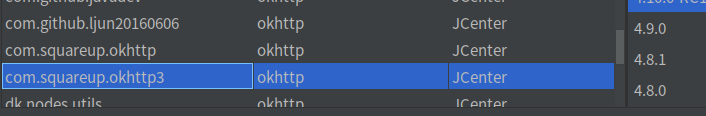
* 在服务端去除其它代码，只留下true或false

**3.3 删除、修改（同样的方法）**

**3.4 OKHttp连接网络**

* 添加库

implementation 'com.squareup.okhttp3:okhttp:4.10.0-RC1'



要选择squareup的okhttp3版本

* 连接网络

*thread* **{** try{  
 val client = OkHttpClient()  
 val request = Request.Builder()  
 .url(book\_all)  
 .build()  
 val response = client.newCall(request).execute()  
 val responseData = response.body?.string()  
 if(responseData != null){  
 showResponse(responseData)  
 }  
 else{  
 showResponse("没有数据")  
 }  
 }  
 catch (e: Exception){  
 e.printStackTrace()  
 }  
**}**

* 发送数据：

val requestBody = FormBody.Builder()  
 .add("name", "新唐书")  
 .add("author", "欧阳修")  
 .add("pages", "1000")  
 .add("price", "120.7")  
 .build()  
val request = Request.Builder()  
 .url(add\_book)  
 .post(requestBody)  
 .build()

如此处理，会有中文乱码问题

乱码问题的解决：

<https://juejin.im/post/6844903861002436616>

通过 UrlEncoder进行编解码，指定utf-8格式

客户端编码指定为utf-8

.add("name", URLEncoder.encode("新唐书", "UTF-8"))

服务端进行解码：

if(name != null && name.length() >0){

name = URLDecoder.decode(name, "utf-8");

}

# 4 实验

将车次表加入到PC机上，完成新增、修改、删除；读取的解析下次完成

# 5 解析xml(对应教材p439)

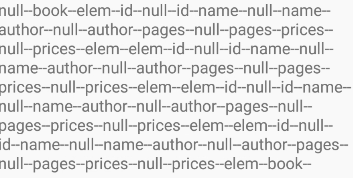
### 5.1 连接网络，只读取静态xml文件

### **5.2 解析xml文件**

1）建立循环，读取节点名称，并用Toast命令读取

// 解析xml  
fun parseXml(xmlStr: String): String{  
 try {  
 val factory = XmlPullParserFactory.newInstance()  
 val xmlPullPaser = factory.newPullParser()  
 // 设置xml解析器的输入  
 xmlPullPaser.setInput(StringReader(xmlStr))  
 var builder = StringBuilder()  
 var eventType = xmlPullPaser.*eventType*// 事件类型  
 while(eventType != XmlPullParser.*END\_DOCUMENT*){  
 val nodeName = xmlPullPaser.*name* builder.append("${nodeName}--")  
 eventType = xmlPullPaser.next() // 读取下一个，进行遍历  
 }  
 runOnUiThread**{** Toast.makeText(this, builder.toString(), Toast.*LENGTH\_LONG*).show()  
 **}** }  
 catch (e: Exception){  
 e.printStackTrace()  
 }  
 return ""  
}

对应的节点名称



解析xml节点内容

START\_TAG, END\_TAG, 以及nextText

while(eventType != XmlPullParser.*END\_DOCUMENT*){  
 val nodeName = xmlPullPaser.*name* // builder.append("${nodeName}--")  
  
 when(eventType){  
 // 标签开始  
 XmlPullParser.*START\_TAG*->{  
 when(nodeName){  
 "id" -> idStr = xmlPullPaser.nextText()  
 "name" -> name = xmlPullPaser.nextText()  
 "author" -> author = xmlPullPaser.nextText()  
 "pages" -> pageStr = xmlPullPaser.nextText()  
 "prices" -> priceStr = xmlPullPaser.nextText()  
  
 }  
 }  
 // 标签结束  
 XmlPullParser.*END\_TAG*->{  
 if("elem" == nodeName){  
 builder.append("$idStr--$name--$author--$pageStr--$priceStr")  
 }  
 }  
 }

Kotlin中的类型转换：

val id = idStr.*toInt*()  
val pages = pageStr.*toInt*()  
val price = priceStr.*toDouble*()

此处应该封装成一个类

### **5.3 与jsp网页连接**

### **5.4 根据条件查询**

**1）查询页数在500到1000之间的**

**2）查询作者姓名中有姓“班”的**

# 6 json

### **6.1 json字符串的格式、生成**

[

{

"id":"2"

, "name":"史记"

, "author":"司马迁"

, "pages":"800"

, "price":"105.3"

}

,

{

"id":"3"

, "name":"三国志"

, "author":"陈寿"

, "pages":"600"

, "price":"85.3"

}

]

### **6.2 jsonobject解析**

try {  
 val jsonArray = JSONArray(responseData)  
 val builder = StringBuilder()  
 for(i in 0 *until* jsonArray.length()){  
 val jsonObject = jsonArray.getJSONObject(i)  
 val id = jsonObject.getInt("id")  
 val name = jsonObject.getString("name")  
 val author = jsonObject.getString("author")  
 val pages = jsonObject.getInt("pages")  
 val price = jsonObject.getDouble("price")  
 builder.append("${id}--${name}--${author}--${pages}--${price}\n")  
 }  
 showResponse(builder.toString())  
}  
catch (e: Exception){  
 e.printStackTrace()  
}

### **6.3 Gson**

导入库：

implementation 'com.google.code.gson:gson:2.8.6'

定义类：

data class BookInfo (val id:Int, val name: String, val author: String, val pages: Int, val price: String)

解析：

private fun parseJsonWithGson(responseData: String){  
 val gson = Gson()  
 val typeOf = object : TypeToken<List<BookInfo>>(){}.*type* val bookList = gson.fromJson<List<BookInfo>>(responseData, typeOf)  
 val builder = StringBuilder()  
 for(info in bookList){  
 builder.append(info.toString()).append("\n")  
 }  
 showResponse(builder.toString())  
}

# 7 第二次实验

通过OKHttp连接网络、修改、删除数据

根据起点站、终点站查询

根据时间段查询

分别使用XML、Json实现（具体功能可以自己设计）

报告要求体现出使用OkHttp的查询、修改、删除；使用xml和json解析

实验内容也可以完成大作业的内容

# 8 网络请求的封装（回调函数）

## 8.1 问题的提出

线程没有返回值（若不使用线程，则造成阻塞）

重复代码过多

object HttpUtil {  
  
 // 发送网络连接  
 fun sendHttpRequest(address: String): String{  
 var connection : HttpURLConnection? = null  
 try {  
 val response = StringBuilder()  
 val url = URL(address)  
 connection = url.openConnection() as HttpURLConnection  
 connection.*connectTimeout* = 8000 // 连接 超时  
 connection.*readTimeout* = 8000 // read 超时时间  
 val input = connection.*inputStream* // 读取输入流，类似与文件流  
 val reader = BufferedReader(InputStreamReader(input))  
 reader.*use***{** reader.*forEachLine* **{** response.append(**it**)  
 **}  
 }** return response.toString()  
 }  
 catch (e: Exception){  
 e.printStackTrace()  
 }  
 finally {  
 connection?.disconnect()  
 }  
  
 }  
}

## 8.2 定义回调接口

interface HttpCallbackListener {  
 // 网络连接正常结束  
 fun onFinish(response: String)  
 // 网络连接发生错误  
 fun onError(e: Exception)  
}

## 8.3 HttpUtil的修改

1）去除返回值

2）新增listener接口参数

3）新增线程

4）网络正常，回调onFinish; 错误：回调onError

// 发送网络连接  
fun sendHttpRequest(address: String, listener: HttpCallbackListener){  
  
 thread **{** var connection: HttpURLConnection? = null  
 try {  
 val response = StringBuilder()  
 val url = URL(address)  
 connection = url.openConnection() as HttpURLConnection  
 connection.*connectTimeout* = 8000 // 连接 超时  
 connection.*readTimeout* = 8000 // read 超时时间  
 val input = connection.*inputStream* // 读取输入流，类似与文件流  
 val reader = BufferedReader(InputStreamReader(input))  
 reader.*use* **{** reader.*forEachLine* **{** response.append(**it**)  
 **}  
 }** // 执行到此，说明网络正常，回调 onFinish函数  
 listener.onFinish(response.toString())  
  
 } catch (e: Exception) {  
 e.printStackTrace()  
 // 出现错误，回调onError方法  
 listener.onError(e)  
 } finally {  
 connection?.disconnect()  
 }  
  
 **}**}

## 8.4 回调函数的应用

调用的参数，需要实现对应的HttpCallbackListener的接口函数

// HttpUrl回调  
btnHttpUrlCallBack.setOnClickListener **{** HttpUtil.sendHttpRequest(book\_all, object: HttpCallbackListener{  
 // 网络正常  
 override fun onFinish(response: String) {  
 showResponse(response)  
 }  
  
 // 出现错误  
 override fun onError(e: Exception) {  
  
 }  
 })  
**}**

同样的，如果要实现新增、删除等需要传递参数的代码，则需要在新函数中，使用connection的post方法，其它的不变

8.5 OkHttp的封装

// 通过OkHttp发送  
fun sendOkHttpRequest(address: String, callback: okhttp3.Callback){  
 val client = OkHttpClient()  
 val request = Request.Builder()  
 .url(address)  
 .build()  
 client.newCall(request).enqueue(callback)  
}

OKHttp已经定义了回调接口okhttp3.Callback

函数中的调用：实现Callback

// OKHttpUtil的封装  
btnOkHttpCallBack.setOnClickListener **{** HttpUtil.sendOkHttpRequest(book\_all, object: Callback{  
 override fun onResponse(call: Call, response: Response) {  
 val responseData = response.body?.string()  
 showResponse(responseData)  
 }  
  
 // 网络失败  
 override fun onFailure(call: Call, e: IOException) {  
 e.printStackTrace()  
 }  
 })  
**}**

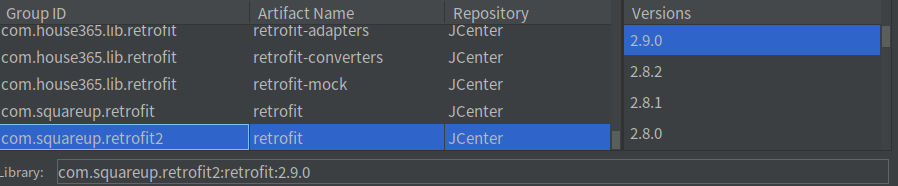
# 9 retrofit的使用

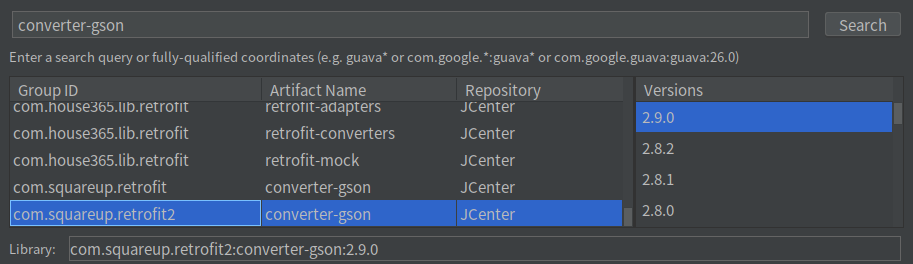
可参考：

<https://blog.csdn.net/carson_ho/article/details/73732076>

<https://segmentfault.com/a/1190000005638577>

## 9.1 导入库





implementation 'com.squareup.retrofit2:retrofit:2.9.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

## 9.2 定义接口

interface BookService {  
 @GET("get\_all\_book\_json.jsp")  
 fun getBookData(): Call<List<BookInfo>>  
}

（BookInfo类在前面已经写好）

## 9.3 使用

private fun readBookByRetrofit(){  
 // 设置基础url  
 val retrofit = Retrofit.Builder()  
 .baseUrl("http://${server\_ip}:8080/TestWeb/book/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()  
 val bookService = retrofit.create(BookService::class.*java*)  
 bookService.getBookData().enqueue(object: retrofit2.Callback<List<BookInfo>> {  
 // 正常  
 override fun onResponse(  
 call: retrofit2.Call<List<BookInfo>>,  
 response: retrofit2.Response<List<BookInfo>>  
 ) {  
 val list = response.body()  
 val builder = StringBuilder()  
 if(list != null) {  
 for(info in list){  
 builder.append(info.toString()).append("\n")  
 }  
 }  
 showResponse(builder.toString())  
 }  
  
 // 失败  
 override fun onFailure(call: retrofit2.Call<List<BookInfo>>, t: Throwable) {  
 t.printStackTrace()  
 }  
 })

编译错误：

# **Error: Invoke-customs are only supported starting with Android O (--min-api 26)**

参照：<https://blog.csdn.net/yinxing2008/article/details/83619313>

compileOptions **{** targetCompatibility 1.8  
 sourceCompatibility 1.8  
**}**

@GET("book/get\_all\_book\_json.jsp")

.baseUrl("http://${server\_ip}:8080/TestWeb/")

这两个相互配合，组合成一个完成的url

## 9.4 传递参数

@GET("book/get\_book\_by\_page\_range\_json.jsp")  
fun getBookByPageRange(@Query("min") min:Int, @Query("max") max: Int): Call<List<BookInfo>>

.getBookByPageRange(100, 1000)

## 9.5 新增POST方法

@FormUrlEncoded  
@POST("book/add\_new\_book.jsp")  
fun addBook(@Field("name") name: String, @Field("author") author: String,  
 @Field("pages") pages: Int, @Field("price") price: Double): Call<ResponseBody>

// 设置基础url  
val retrofit = Retrofit.Builder()  
 .baseUrl("http://${server\_ip}:8080/TestWeb/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()  
val bookService = retrofit.create(BookService::class.*java*)  
bookService  
 .addBook(URLEncoder.encode("新唐书", "UTF-8"),  
 URLEncoder.encode("欧阳修", "UTF-8"), 1200, 305.6)  
 .enqueue(object: retrofit2.Callback<ResponseBody> {  
 override fun onResponse(  
 call: retrofit2.Call<ResponseBody>,  
 response: retrofit2.Response<ResponseBody>  
 ) {  
 showResponse("新增成功")  
 }  
  
 override fun onFailure(call: retrofit2.Call<ResponseBody>, t: Throwable) {  
 t.printStackTrace()  
 }  
 })