- update：更新数据。
- delete：删除数据。
- getType：获取数据类型。

这些方法看起来是不是很像 SQLite？没错，ContentProvider 作为中间的接口，本身并不直接保存数据，而是通过 SQLiteOpenHelper 与 SQLiteDatabase 间接操作底层的 SQLite。所以要想使用 ContentProvider，首先得实现 SQLite 的数据表帮助器，再由 ContentProvider 封装对外的接口。

下面是使用 ContentProvider 提供用户信息对外接口的 Kotlin 代码例子：

```
    class UserInfoProvider : ContentProvider() {
        lateinit var userDB: UserDBHelper

        //删除数据
        override fun delete(uri: Uri, selection: String?, selectionArgs:
Array<String>?): Int {
            var count = 0
            if (uriMatcher.match(uri) == USER_INFO) {
                count = userDB.delete(selection, selectionArgs)
            }
            return count
        }

        //插入数据
        override fun insert(uri: Uri, values: ContentValues?): Uri? {
            var newUri = uri
            if (uriMatcher.match(uri) == USER_INFO) {
                // 向指定的表插入数据，得到返回的 Id
                val rowId = userDB.insert(values)
                if (rowId > 0) { // 判断插入是否执行成功
                    // 如果添加成功，就利用新添加的 Id 和生成的新地址
                    newUri = ContentUris.withAppendedId
(UserInfoContent.CONTENT_URI, rowId)
                    // 通知监听器，数据已经改变
                    context.contentResolver.notifyChange(newUri, null)
                }
            }
            return newUri
        }

        //创建 ContentProvider 时调用的回调函数
        override fun onCreate(): Boolean {
            userDB = UserDBHelper.getInstance(context, 1)
            return false
        }
```

```
        //查询数据库
        override fun query(uri: Uri, projection: Array<String>?, selection:
String?,
                           selectionArgs: Array<String>?, sortOrder: String?):
Cursor? {
            var cursor: Cursor? = null
            if (uriMatcher.match(uri) == USER_INFO) {
                // 执行查询
                cursor = userDB.query(projection, selection, selectionArgs,
sortOrder)
                // 设置监听
                cursor?.setNotificationUri(context.contentResolver, uri)
            }
            return cursor

        }

        //获取数据访问类型，暂未实现
        override fun getType(uri: Uri): String? {
            throw UnsupportedOperationException("Not yet implemented")
        }

        //更新数据，暂未实现
        override fun update(uri: Uri, values: ContentValues?, selection: String?,
selectionArgs: Array<String>?): Int {
            throw UnsupportedOperationException("Not yet implemented")
        }

        companion object {
            val USER_INFO = 1
            val uriMatcher = UriMatcher(UriMatcher.NO_MATCH)
            //伴生对象的初始化操作
            init {
                uriMatcher.addURI(UserInfoContent.AUTHORITIES, "/user",
USER_INFO)
            }
        }
    }
```

既然内容提供器是四大组件之一，就得在 AndroidManifest.xml 中注册它的定义，并对其开放外部访问的权限，注册代码示例如下：

```
            <!-- 注册用户信息的内容提供器 -->
            <provider
                android:name=".provider.UserInfoProvider"
                android:authorities=
```

```
"com.example.network.provider.UserInfoProvider"
                android:enabled="true"
                android:exported="true" />
```

provider 注册完毕，如此便完成了服务端 App 的封装工作，接下来即可由其他 App 进行数据存取。

10.4.2 内容解析器 ContentResolver

10.4.1 小节提到了利用 ContentProvider 实现服务端 App 的数据封装，如果其他 App 想访问服务端 App 的内部数据，就要通过内容解析器 ContentResolver 来访问。内容解析器是外部 App 操作服务端数据的工具，相对应的内容提供器是服务端的数据接口。若要获取 ContentResolver 对象，则可在 Activity 代码中调用 getContentResolver 方法，现在 Kotlin 中可直接使用属性 ContentResolver 取代方法 getContentResolver()。

ContentResolver 提供的方法与 ContentProvider 是一一对应的，比如 query、insert、update、delete、getType 等方法，连方法的参数类型都一模一样。其中，最常用的是 query 方法，调用该方法返回一个游标 Cursor 对象，这个游标与 SQLite 的游标是同样的，想必读者早已用得炉火纯青。

下面是 query 方法的具体参数说明。

- uri：Uri 类型，可以理解为本次操作的数据表路径。
- projection：字符串数组类型，指定将要查询的字段名称列表。
- selection：字符串类型，指定查询条件。
- selectionArgs：字符串数组类型，指定查询条件中的参数取值列表。
- sortOrder：字符串类型，指定排序条件。

针对 10.4.1 小节 UserInfoProvider 提供的数据接口，下面是使用 ContentResolver 在外部 App 添加用户信息的 Kotlin 代码例子：

```
    private fun addUser(resolver: ContentResolver, user: UserData) {
        val name = ContentValues()
        name.put("name", user.name)
        name.put("age", user.age)
        name.put("height", user.height)
        name.put("weight", user.weight)
        name.put("married", false)
        name.put("update_time", DateUtil.getFormatTime())
        //UserInfoContent.CONTENT_URI 指向的字符串就是 provider 在
AndroidManifest.xml 里的 android:authorities 属性值
        resolver.insert(UserInfoContent.CONTENT_URI, name)
    }
```

下面是使用 ContentResolver 在客户端查询所有用户信息的代码例子：

```
    private fun readAllUser(resolver: ContentResolver): String {
        val userArray = ArrayList<UserData>()
```

```
            val cursor = resolver.query(UserInfoContent.CONTENT_URI, null, null,
null, null)
            while (cursor.moveToNext()) {
                val user = UserData()
                user.name = cursor.getString(cursor.getColumnIndex
(UserInfoContent.USER_NAME))
                user.age = cursor.getInt(cursor.getColumnIndex
(UserInfoContent.USER_AGE))
                user.height = cursor.getInt(cursor.getColumnIndex
(UserInfoContent.USER_HEIGHT)).toLong()
                user.weight = cursor.getFloat(cursor.getColumnIndex
(UserInfoContent.USER_WEIGHT))
                userArray.add(user)
            }
            cursor.close()
            var result = ""
            for (user in userArray) {
                result = "$result${user.name}  年龄${user.age}  身高${user.height}
体重${user.weight}\n"
            }
            return result
        }
```

利用内容解析器添加用户信息的效果如图 10-20 所示，一开始服务端的用户表不存在用户记录，外部 App 通过 ContentResolver 添加一条记录后，服务端的用户记录数返回 1。用户信息的查询明细如图 10-21 所示，点击页面上的用户记录数量，弹出一个对话框，提示当前找到的所有用户的明细数据，包括姓名、年龄、身高、体重等信息。

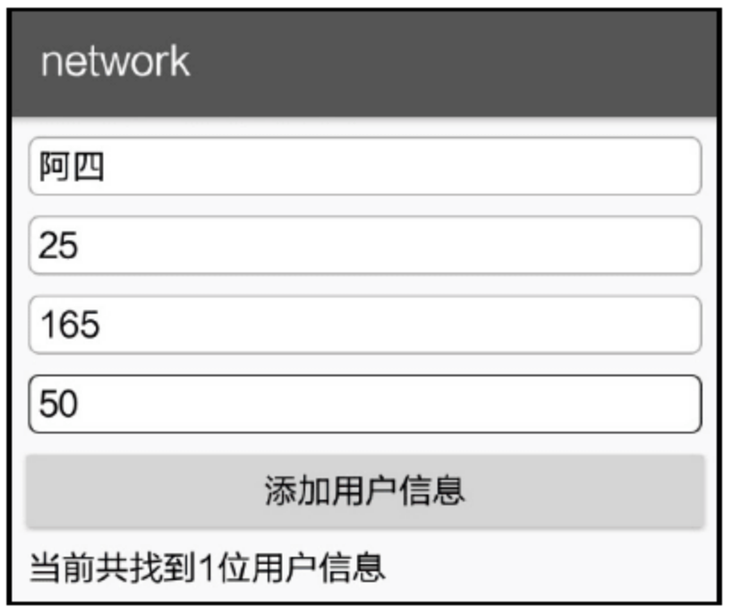

图 10-20　添加一条用户信息的界面

图 10-21　用户信息明细的查询结果

实际开发中，普通 App 很少会开放数据接口给其他应用访问，所以作为服务端接口的 ContentProvider 反而基本用不到。内容组件能够派上用场的情况往往是 App 想要访问系统应用的通信数据，比如查看联系人、短信、通话记录以及对这些通信信息进行增删改查。

下面是使用 ContentResolver 添加联系人信息的 Kotlin 代码片段，此时访问的数据来源变成了系统自带的联系人资源库“content://com.android.contacts/”：

```
        fun addContacts(resolver: ContentResolver, contact: Contact) {
            val raw_uri = Uri.parse("content://com.android.contacts/
raw_contacts")
            val values = ContentValues()
            // 添加一条联系人的主记录，并返回唯一的联系人编号
            val contactId = ContentUris.parseId(resolver.insert(raw_uri, values))
            val uri = Uri.parse("content://com.android.contacts/data")
            // 添加联系人姓名（要根据前面获取的 id 号）
            val name = ContentValues()
            name.put("raw_contact_id", contactId)
            name.put("mimetype", "vnd.android.cursor.item/name")
            name.put("data2", contact.name)
            resolver.insert(uri, name)
            // 添加联系人的手机号码
            val phone = ContentValues()
            phone.put("raw_contact_id", contactId)
            phone.put("mimetype", "vnd.android.cursor.item/phone_v2")
            phone.put("data2", "2")
            phone.put("data1", contact.phone)
            resolver.insert(uri, phone)
            // 添加联系人的电子邮箱
            val email = ContentValues()
            email.put("raw_contact_id", contactId)
            email.put("mimetype", "vnd.android.cursor.item/email_v2")
            email.put("data2", "2")
            email.put("data1", contact.email)
            resolver.insert(uri, email)
        }
```

注意到上述代码用了 4 条 insert 语句，但业务上只添加了一个联系人信息。这样处理有个问题，就是 4 个 insert 操作不在同一个事务中，要是中间某步 insert 操作失败，那么之前插入成功的记录无法自动回滚，从而产生垃圾数据。

为了避免这种情况的发生，Android 又提供了内容操作器 ContentProviderOperation 进行批量数据的处理。内容操作器在一个请求中封装多条记录的修改动作，然后一次性提交给服务端，这就实现了在一个事务中完成多条数据的更新操作。即使某条记录处理失败，ContentProviderOperation 也能根据事务一致性原则，自动回滚本事务已经执行的修改操作。

下面是使用 ContentProviderOperation 批量添加联系人信息的 Kotlin 代码片段：

```
        fun addFullContacts(resolver: ContentResolver, contact: Contact) {
            val raw_uri = Uri.parse("content://com.android.contacts/
raw_contacts")
            val uri = Uri.parse("content://com.android.contacts/data")
            // 依次封装联系人主记录、联系人姓名、手机号码、电子邮箱的操作行为
            val op_main = ContentProviderOperation
                    .newInsert(raw_uri).withValue("account_name", null).build()
```

```
        val op_name = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/name")
                .withValue("data2", contact.name).build()
        val op_phone = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/phone_v2")
                .withValue("data2", "2").withValue("data1", contact.phone)
                .build()
        val op_email = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/email_v2")
                .withValue("data2", "2").withValue("data1", contact.email)
                .build()
        // 把以上 4 个操作行为组成行为队列，并一次性处理解决该行为队列
        val operations = mutableListOf(op_main, op_name, op_phone, op_email)
        resolver.applyBatch("com.android.contacts", operations as
ArrayList<ContentProviderOperation>)
    }
```

采取批量方式添加联系人信息的效果如图 10-22 和图 10-23 所示，其中图 10-22 所示为添加之前的截图，此时联系人个数为 174 位；图 10-23 所示为添加成功之后的截图，此时联系人个数为 175 位。

图 10-22　添加联系人之前的截图

图 10-23　添加联系人之后的截图

10.4.3　内容观察器 ContentObserver

ContentResolver 获取外部数据采用的是主动查询方式，有去查就有数据，没去查就没数据。但有时 App 不但要获取以往的数据，还要实时获取新增的数据，最常见的业务场景便是短信验证码。电商 App 经常在用户注册或者付款时下发验证码短信，为了帮用户省事，App 通常会监控手机刚收到的验证码数字，并自动填入验证码输入框。这时就用到了内容观察器 ContentObserver，它给目标内容注册一个观察器，然后目标内容的数据一旦发生变化，就马上触发观察器规定好的动作，从而执行开发者预先定义的代码。

内容观察器的用法与内容提供器类似，也要从 ContentObserver 派生出一个观察器类，然后通过 ContentResolver 对象调用相应的方法注册或注销观察器。ContentResolver 与观察器有关的方法说明如下。

- registerContentObserver：注册内容观察器。
- unregisterContentObserver：注销内容观察器。
- notifyChange：通知内容观察器发生了数据变化。

为了让读者能够更好地理解，还是举个实际应用的例子。很多手机安全 App 都具备流量校准的功能，只要把特定格式的短信发送给移动运营商，就会收到运营商下发的流量校准短信，通过解析这个流量短信，即可获取详细的用户流量数据，包括月流量额度、已使用流量、未使用流量等信息。

以中国移动的手机号码为例，发送短信内容“18”到客服号码 10086，不一会儿便会收到 10086 发来的流量结果短信。下面是利用 ContentObserver 实现流量校准功能的 Kotlin 页面代码：

```
class ContentObserverActivity : AppCompatActivity() {
    private var mObserver: SmsGetObserver? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_content_observer)
        Observer.tv_check_flow = findViewById<TextView>(R.id.tv_check_flow)
        tv_check_flow.setOnClickListener {
            alert(mCheckResult, "收到流量校准短信") {
                positiveButton("确定") {}
            }.show()
        }
        btn_check_flow.setOnClickListener {
            var dialog = indeterminateProgressDialog("正在进行流量校准", "请稍候")
            dialog.show()
            //查询数据流量，移动号码的查询方式为发送短信内容“18”给“10086”
            //电信和联通号码的短信查询方式请咨询当地运营商客服热线
            sendSmsAuto("10086", "18")
            Handler().postDelayed({
                if (dialog.isShowing == true) {
                    dialog.dismiss()
                }}, 5000)
        }
        mSmsUri = Uri.parse("content://sms")
        mSmsColumn = arrayOf("address", "body", "date")
        mObserver = SmsGetObserver(this, Handler())
        //注册短信接收的内容观察器
        contentResolver.registerContentObserver(mSmsUri!!, true, mObserver!!)
    }

    override fun onDestroy() {
        //注销短信接收的内容观察器
```

```
            contentResolver.unregisterContentObserver(mObserver!!)
            super.onDestroy()
        }

        //短信发送广播，如需处理可注册该事件的 BroadcastReceiver
        private val SENT_SMS_ACTION = "com.example.network.SENT_SMS_ACTION"
        //短信接收广播，如需处理可注册该事件的 BroadcastReceiver
        private val DELIVERED_SMS_ACTION = "com.example.network.
DELIVERED_SMS_ACTION"

        fun sendSmsAuto(phoneNumber: String, message: String) {
            //声明短信发送的广播意图
            val sentIntent = Intent(SENT_SMS_ACTION)
            sentIntent.putExtra("phone", phoneNumber)
            sentIntent.putExtra("message", message)
            val sentPI = PendingIntent.getBroadcast(this, 0, sentIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //声明短信接收的广播意图
            val deliverIntent = Intent(DELIVERED_SMS_ACTION)
            deliverIntent.putExtra("phone", phoneNumber)
            deliverIntent.putExtra("message", message)
            val deliverPI = PendingIntent.getBroadcast(this, 1, deliverIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //要确保打开发送短信的完全权限，不是那种还需提示的不完整权限
            val smsManager = SmsManager.getDefault()
            smsManager.sendTextMessage(phoneNumber, null, message, sentPI,
deliverPI)
        }

        //定义一个短信观察器的嵌套类
        private class SmsGetObserver(private val mContext: Context, handler:
Handler) : ContentObserver(handler) {

            override fun onChange(selfChange: Boolean) {
                var sender = ""
                var content = ""
                //查询收件箱中来自 10086 的最近短信
                val selection = "address='10086' and
date>${System.currentTimeMillis()-1000*60*60}"
                val cursor = mContext.contentResolver.query(
                        mSmsUri!!, mSmsColumn, selection, null, " date desc")
                while (cursor.moveToNext()) {
                    sender = cursor.getString(0)
                    content = cursor.getString(1)
                    break
                }
                cursor.close()
```

```
            mCheckResult = "发送号码：$sender\n 短信内容：$content"
            //将解析后的短信内容显示到界面上
            Observer.tv_check_flow!!.text = "流量校准结果如下：\n\t" +
                    "总流量为：${findFlow(content, "总流量为", "MB")}\n\t" +
                    "已使用：${findFlow(content, "已使用", "MB")}\n\t" +
                    "剩余：${findFlow(content, "剩余", "MB")}"
            super.onChange(selfChange)
        }
    }

    companion object Observer {
        private var tv_check_flow: TextView? = null
        private var mCheckResult: String = ""
        private var mSmsUri: Uri? = null
        private var mSmsColumn: Array<String>? = null
        //在伴生对象中定义解析短信内容的方法
        private fun findFlow(sms: String, begin: String, end: String): String {
            val begin_pos = sms.indexOf(begin)
            if (begin_pos < 0) {
                return "未获取"
            }
            val sub_sms = sms.substring(begin_pos)
            val end_pos = sub_sms.indexOf(end)
            return if (end_pos < 0) {
                "未获取"
            } else sub_sms.substring(begin.length, end_pos + end.length)
        }
    }
}
```

根据运营商短信进行流量校准的效果如图 10-24 和图 10-25 所示，其中图 10-24 所示为用户实际收到的短信内容，图 10-25 所示为 App 监视短信并解析完成的流量数据页面。

图 10-24　用户收到的短信内容

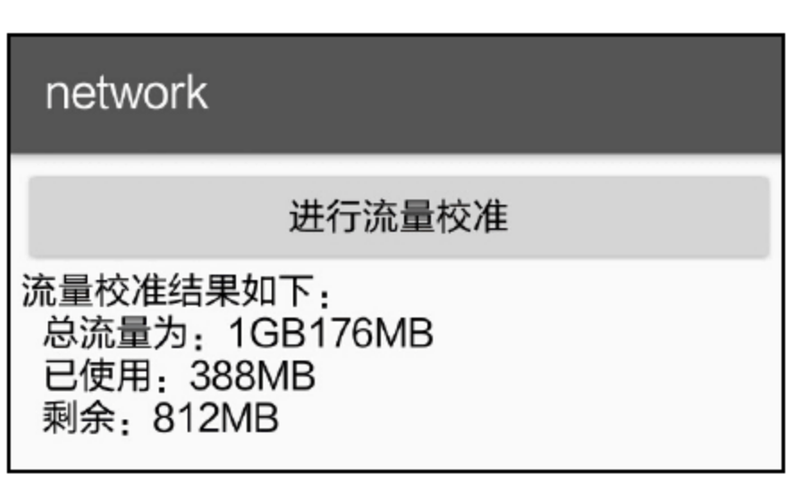

图 10-25　App 解析后的流量数据

10.5　实战项目：电商 App 的自动升级

都说酒香不怕巷子深，又说养在深闺人未识，这两句话看似矛盾，其实指的是同一个含义，即美好的事物要想办法让大众了解才会闻名。无论是香飘十里，还是抛头露面，目的都是开展营销，也就是俗话说的做广告。开发电商 App 也是如此，要想让大家知道这里的商品质量过硬、价格公道，势必经常举办各种促销活动，比如一年一度的 618、双十一等。既然频繁搞活动，就得进行技术支撑，隔三岔五对 App 升级一下，增加几个新功能，修复一些老 BUG 等。这个 App 升级功能便是本章末尾的实战项目“电商 App 的自动升级”，这个自动升级到底用到了哪些技术？下面就来看看该项目的分析与设计。

10.5.1　需求描述

要说安装 App，可能有的小伙伴觉得那还不简单，直接找个应用商店搜索 App 名称不就行了？但初次安装与安装后升级不是一个概念，如果升级也靠用户手动到应用商店搜索并安装，那就“图样图森破”了。贴心的做法是在 App 内部提供在线更新的功能，这个在线更新又分为两种形式，一种是由用户手工选择应用菜单上的“检查更新”，另一种是 App 启动后自行判断服务器上是否有更高版本的安装包。“检查更新”的菜单项位置在每个 App 内部都不一样，App 自动进行升级判断则后台服务并没有对应的界面，所以在线更新的效果图暂且按照图 10-26 所示的样子，效果图上有两个按钮，其中“不请求接口直接弹窗”按钮模拟了用户点击“检查更新”菜单的动作，“请求服务端接口再弹窗”按钮模拟的则是 App 自动检查升级的功能。

无论是手动更新还是自动更新，结果都要调用后端服务器的版本更新接口，根据服务器的应答报文判断是否需要升级。如果服务器返回有更新的安装包，App 界面就弹窗提示用户要不要在线升级，提示窗如图 10-27 所示。注意这里有种特殊的情况，倘若 App 扫描设备发现存储卡上已经存在新版本的 APK 安装包，则提示用户本地已经有了新包，无须耗费流量即可进行升级，此时的提示窗如图 10-28 所示。

图 10-26　商城首页模拟两种升级动作

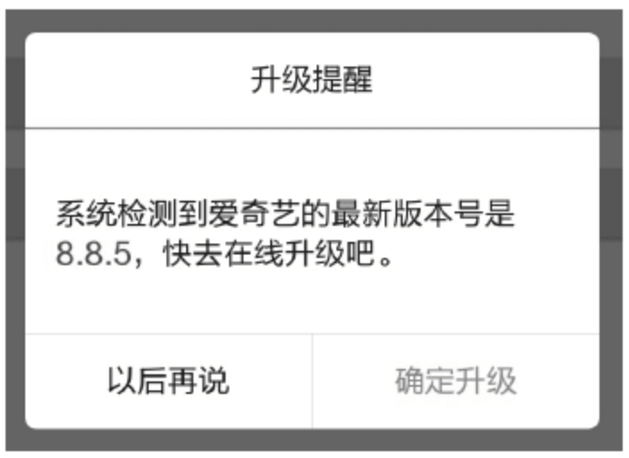

图 10-27　需要下载安装包的提示窗

图 10-28　无须下载安装包的提示窗

对于手机存储卡未找到新版本 APK 的情况，只要用户在提示框中选择“确定升级”，App 就

要一边下载 APK 文件，一边弹出进度对话框，包含下载进度的对话框如图 10-29 所示。APK 安装包下载完毕，或者 App 发现手机原来已有对应版本的安装包，接着用户确认升级操作之后，App 应当继续 APK 的安装动作，安装期间仍需弹窗提示正在安装，安装提示对话框如图 10-30 所示。

等待 App 完成新版本的升级，回到升级前的 App 界面，然后弹出升级完毕的提示对话框，如图 10-31 所示，至此方才结束 App 的自动升级历程。

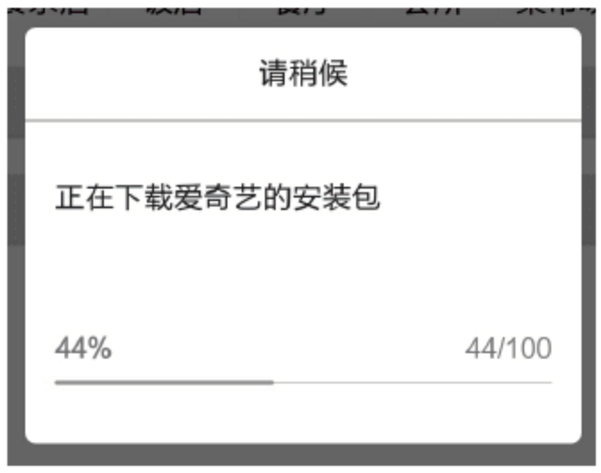

图 10-29　下载过程中的进度对话框

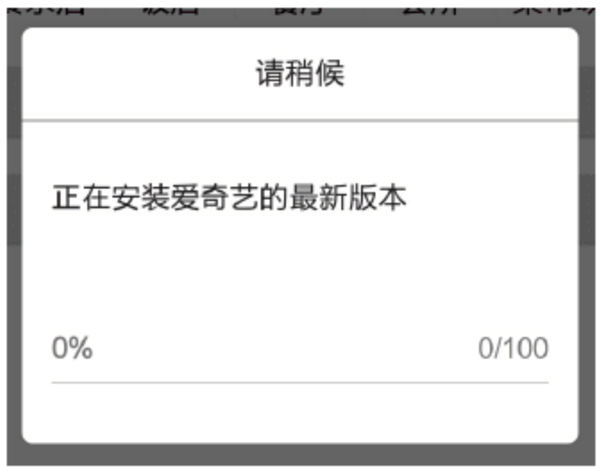

图 10-30　安装过程中的进度对话框

图 10-31　升级完毕的提示对话框

另外，上面几个提示对话框中的消息文本需要高亮显示升级后的最新版本号，从而更好地提醒用户正在升级的是哪个版本。

10.5.2　开始热身：可变字符串 SpannableString

10.5.1 小节的需求描述说到提示框中的最新版本要用高亮文字显示，可是文本视图只提供了统一的文本样式设置方法，比如通过 setTextColor 方法设置文本颜色，通过 setTextSize 方法设置文本大小，还可以通过 setTextAppearance 方法设置文本风格（包括颜色、大小、对齐方式等）。一旦调用了这些方法，文本内容就会显示同样的颜色、同样的大小或者同样的风格，根本没法让内部某些文字单独高亮显示。因此，为了解决分段文本展示不同样式的需求，Android 提供了可变字符串 SpannableString，通过该工具实现对文本样式的分段显示。

可变字符串的原理是给指定位置的文本赋予对应的样式，从而告知系统这段文本的显示方式。具体到 Kotlin 编码上，主要有三个步骤，说明如下。

步骤 01　从指定文本字符串构造一个 SpannableString 对象。

步骤 02　调用 SpannableString 对象的 setSpan 方法设置指定文本段的显示风格。该方法的第一个参数为风格样式，第二个参数为文本段的起始位置，第三个参数为文本段的终止位置，第四个参数为风格的范围标志（一般设置为 Spanned.SPAN_EXCLUSIVE_EXCLUSIVE）。

步骤 03　把处理好的 SpannableString 对象赋值给文本视图的 text 属性。注意 text 字段的类型并非 String，而是 CharSequence，因此凡是实现了 CharSequence 接口的类，其对象都允许赋值给 text 字段。字符串 String 类当然有实现 CharSequence，同样可变字符串 SpannableString 类也实现了该接口，故而它们的实例均为合法的 text 参数。

由此可见，运用可变字符串的关键是第二步，不过第二步的 setSpan 方法每次也只能单独设置一处的文字样式，倘若原字符串有多处文本需要定制文字样式，那还得多次调用 setSpan 方法，这样盘算依然不太经济。好在强大的 Anko 库又封装了一个扩展函数 buildSpanned，只要在该函数内

部调用形如“append(待定制样式的文本, 定制后的风格样式)”的代码，即可返回自动拼接后的多彩文本串。

下面是通过 buildSpanned 函数连续构建可变字符串的 Kotlin 代码例子：

```
val str: Spanned = buildSpanned {
    append("为", StyleSpan(Typeface.BOLD)) //文字字体使用粗体
    append("人民", RelativeSizeSpan(1.5f)) //文字大小增大到 1.5 倍大
    append("服务", ForegroundColorSpan(Color.RED)) //文字颜色使用红色
    append("是谁", BackgroundColorSpan(Color.GREEN)) //背景色使用绿色
    append("提出来的", UnderlineSpan()) //文字下方增加下划线
}
```

上面的 Kotlin 眼瞅着还算整齐，然而仅仅为了表示粗体就得书写完整的风格声明代码“StyleSpan(Typeface.BOLD)”着实不够干脆。所以 Anko 索性在这里快刀斩乱麻，又把“StyleSpan(Typeface.BOLD)”简化成了“Bold”，其他几个风格声明也陆续予以缩写。于是最终简化后的 Kotlin 代码如下所示：

```
val str: Spanned = buildSpanned {
    append("为", Bold) //文字字体使用粗体
    append("人民", RelativeSizeSpan(1.5f)) //文字大小增大到 1.5 倍大
    append("服务", foregroundColor(Color.RED)) //文字颜色使用红色
    append("是谁", backgroundColor(Color.GREEN)) //背景色使用绿色
    append("提出来的", Underline) //文字下方增加下划线
}
```

以上构建可变字符串用到的风格样式只是沧海一粟，完整的风格样式定义在 android.text.style 包中，总共有 30 多个类型，当然常用的没这么多，笔者整理了几个常用的风格样式，结合 Anko 库的简化写法，具体说明见表 10-5。

表 10-5　常用的显示风格列表与 Anko 库的简化写法

可变字符串的显示风格类	Anko 库的简化写法	说明
RelativeSizeSpan	无	设置文字的相对大小
StyleSpan(Typeface.BOLD)	Bold	设置粗体文字
StyleSpan(Typeface.Italic)	Italic	设置斜体文字
ForegroundColorSpan	foregroundColor	设置文字的颜色
BackgroundColorSpan	backgroundColor	设置文字的背景色
UnderlineSpan	Underline	给文字加上下划线
StrikethroughSpan	Strikethrough	给文字加上删除线
ImageSpan	无	把文字替换为图片

接着对可变字符串分别运用不同的文字样式，看看到底都有哪些风格的显示效果。下面是使用可变字符串设置文字样式的 Kotlin 代码例子：

```
    class SpannableActivity : AppCompatActivity() {
        private val spannables = listOf("增大字号", "加粗字体", "前景红色",
"背景绿色", "下划线", "表情图片", "Anko 自定义")
        private val text = "为人民服务"
        private val key = "人民"
        private var beginPos = text.indexOf(key)
        private var endPos = beginPos + key.length

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_spannable)
        tv_spannable.text = text
        tv_spinner.text = spannables[0]
        tv_spinner.setOnClickListener {
            selector("请选择可变字符串样式", spannables) { i ->
                tv_spinner.text = spannables[i]
                val spanText = SpannableString(text)
                //对这段文本运用指定的风格样式
                spanText.setSpan(when (i) {
                    0 -> RelativeSizeSpan(1.5f) //文字大小增大到 1.5 倍大
                    1 -> StyleSpan(Typeface.BOLD) //文字字体使用粗体
                    2 -> ForegroundColorSpan(Color.RED) //文字颜色使用红色
                    3 -> BackgroundColorSpan(Color.GREEN) //背景色使用绿色
                    4 -> UnderlineSpan() //文字下方增加下划线
                    else -> ImageSpan(this@SpannableActivity, R.drawable.people)
//把文字替换为图片
                    }, beginPos, endPos, Spanned.SPAN_EXCLUSIVE_EXCLUSIVE)
                tv_spannable.text = spanText
                if (i >= 6) {
                    //使用 Anko 库的 buildSpanned 函数连续构建可变字符串
                    tv_spannable.text = buildSpanned {
                        append("为", Bold)
                        append("人民", RelativeSizeSpan(1.5f))
                        append("服务", foregroundColor(Color.RED))
                        append("是谁", backgroundColor(Color.GREEN))
                        append("提出来的", Underline)
                    }
                }
            }
        }
    }
```

由于上面的 Kotlin 代码用到了 Anko 库的扩展函数（包括 buildSpanned、append 等），所以务必在代码头部加入下面一行导入语句：

```
import org.jetbrains.anko.*
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

可变字符串所呈现的不同风格效果如图 10-32～图 10-38 所示，其中图 10-32 所示为加大字体后的效果，图 10-33 所示为加粗字体后的效果，图 10-34 所示为修改文字颜色后的效果，图 10-35 所示为修改文字背景后的效果，图 10-36 所示为增加下画线后的效果，图 10-37 所示为把文字替换成图片后的效果，图 10-38 所示为采用 Anko 函数混合多种样式后的效果。

图 10-32　加大字体的字符串效果

图 10-33　加粗字体的字符串效果

图 10-34　变更文字颜色的字符串效果

图 10-35　变更背景颜色的字符串效果

图 10-36　添加下划线的字符串效果

图 10-37　文字换图片的字符串效果

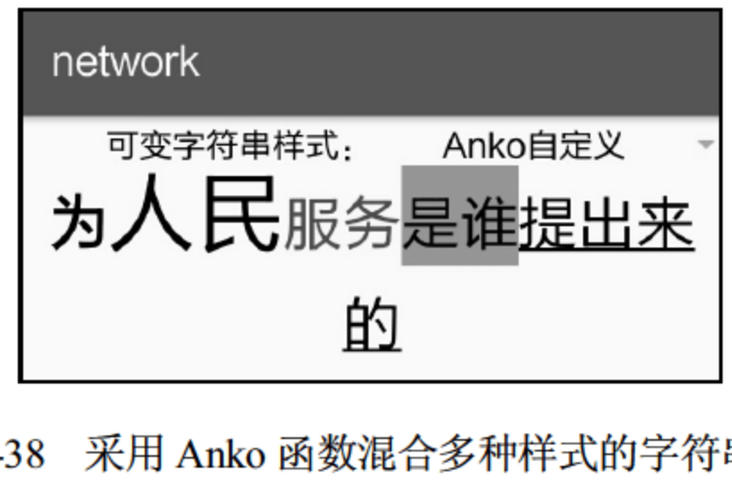

图 10-38　采用 Anko 函数混合多种样式的字符串效果

10.5.3　控件设计

由于自动升级功能更多是在后台完成的，界面上用到的控件反而不多，因此下面罗列的并不限于控件，还包括后台的处理技术。

（1）提醒对话框 AlertDialog：用于提醒用户是否立即升级应用，以及升级完毕之后的提示语。

（2）进度对话框 ProgressDialog：在下载 APK 文件的时候，以及 APK 安装过程中，都要显示进度对话框。

（3）可变字符串 SpannableString：提示文本中需要将最新版本号高亮显示，这便用到了可变字符串。

（4）多线程：App 与后端服务器进行接口交互，需要开启分线程才能调用 HTTP 接口。

（5）网络地址 URL：Kotlin 对 URL 类进行了扩展，增加了 readText 方法用于获取接口数据。

（6）移动数据格式 JSON：后端接口返回 JSON 格式的版本升级信息字符串，然后 App 把 JSON 串转换为数据类对象。

（7）下载管理器 DownloadManager：用于 APK 文件的下载行为，包括下载进度的查询操作。

（8）应用包管理器 PackageManager：根据应用包管理器获得应用的版本号信息。

除了上面提到的技术以外，也会用到内容解析器 ContentResolver，因为需求提到：如果手机内存能找到最新版本的安装包，那就无须下载直接安装即可。若要查找手机上的 APK 安装包，可到媒体资源库中查询媒体类型为“application/vnd.android.package-archive”的文件，该类型表示安卓应用的安装包，也就是 APK 文件。

10.5.4 关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成自动升级项目，下面列举几个重要功能的 Kotlin 代码片段：

1. 关于向服务器请求版本更新信息

访问后端的 HTTP 接口，倘若使用 Java 编码，必定又是长篇大论。采用 Kotlin 编码的瘦身效果立竿见影，只要通过 doAsync+uiThread 组合，接口调用的操作就变得轻描淡写。下面是 HTTP 接口访问的 Kotlin 代码例子：

```
        btn_need_request.setOnClickListener {
            val pi = packageManager.getPackageInfo(packageName, 0)
            //开启分线程执行后端接口调用
            doAsync {
                //从服务端获取版本升级信息
                val url =
"$checkUrl?package_name=${pi.packageName}&version_name=${pi.versionName}"
                val result = URL(url).readText()
                //回到主线程在界面上弹窗提示待升级的版本
                uiThread { checkUpdate(result) }
            }
        }
```

既然是访问服务器的接口，肯定要有对应的服务端程序，这个服务端程序可见本书源代码中的 HttpTest 工程包。

2. 关于高亮显示一段文本中的指定文字

在已有的文本中高亮显示其中的某些文字，处理起来还有点烦琐，首先要找到指定文字在整段文本中的位置，再对这些文字设置对应的高亮样式。鉴于高亮功能比较通用，因此可以写到工具类里面，不过既然使用 Kotlin 编码，建议考虑采取扩展函数的形式将高亮处理函数作为 String 类的一个扩展函数，这样用起来更加方便。

下面是对 String 类进行扩展、添加 highlight 高亮函数的 Kotlin 代码：

```
    //字符串中的关键语句用指定样式高亮显示
    fun String.highlight(key: String, style: CharacterStyle): SpannableString {
        val spanText = SpannableString(this)
        val beginPos = this.indexOf(key)
        val endPos = beginPos + key.length
        spanText.setSpan(style, beginPos, endPos,
Spanned.SPAN_EXCLUSIVE_EXCLUSIVE)
        return spanText
    }
```

有了上述的扩展函数定义，外部就能直接调用字符串对象的 highlight 方法，完全无须记忆工具类的名称。具体的 Kotlin 调用代码如下所示：

```
    val spanText = message.highlight(vc.version_name,
ForegroundColorSpan(Color.RED))
```

同时不要忘了在代码文件头部添加下面的一行导入语句，表示此处用到了自己扩展的 highlight 方法：

```
    import com.example.network.util.highlight
```

3. 关于 alert 如何显示可变字符串的文本内容

前面刚设置好可变字符串的文本内容，不料发现无法利用 alert 方法显示该文本了，怎么回事？这是因为在 Java 编码中，AlertDialog.Builder 的 setTitle 和 setMessage 两个方法的输入参数都是 CharSequence，自然允许将可变字符串对象赋值进去。然而 Anko 库扩展出来的 alert 方法，message 和 title 这两个入参类型却改成 String 字符串了，使得 SpannableString 与 String 类型不同，因此无法输入。

没想到还有这样的事情，真叫“攻城狮”颜面何在。不过作为程序员可得不畏艰难险阻，逢山开路、遇水搭桥，现在 Anko 库造的 alert 桥过不了，不妨自己搭个自定义的新桥，偷梁换柱，把 message 字段的参数类型改成 CharSequence 就行了。于是改写后的 alert 方法代码就变成下面这样了：

```
    //Anko 自带的 alert 只支持 String 类型的文本，不支持富文本的 CharSequence 类型
    //故此处重写 alert 方法，使之支持可变字符串 SpannableString
    fun Context.alert(
            message: CharSequence,
            title: String? = null,
            init: (AlertDialogBuilder.() -> Unit)? = null
    ) = AlertDialogBuilder(this).apply {
        if (title != null) title(title)
```

```
        message(message)
        if (init != null) init()
    }
```

改写完毕，不要忘了在 Activity 代码头部添加下面的一行导入语句，表示本页面用的是自定义的 alert 方法：

```
    import com.example.network.util.alert
```

4. 关于如何获取手机上的 APK 文件

前面“10.5.3 控件设计”小节提到，利用内容解析器 ContentResolver 可到资源库中查询媒体类型为“application/vnd.android.package-archive”的 APK 文件，但这只是大概的思路。因为即便找到了几个 APK 文件，App 又如何甄别这些 APK 都是什么来头、哪个 APK 文件才符合当前应用的指定版本号呢？所以要想逐个判定 APK 文件的真实身份，还得解析 APK 内部的包信息，具体的工作则是调用 PackageManager 对象的 getPackageArchiveInfo 方法，该方法可从指定的 APK 路径获取安装包的详细数据，包括应用的包名、应用的版本号等。有了包名、版本号这些信息，App 方能鉴定本地是否存在最新版本的升级包。

下面是获取并校验本设备上 APK 文件的 Kotlin 代码例子：

```
        private fun getLocalPath(vc: VersionCheck): String {
            var local_path = ""
            //遍历本地所有的 apk 文件
            val cursor = contentResolver.query(MediaStore.Files.getContentUri
("external"),
                    null, "mime_type=\"application/vnd.android.package-archive\"",
null, null)
            while (cursor.moveToNext()) {
                //TITLE 获取文件名，DATA 获取文件完整路径，SIZE 获取文件大小
                val path = cursor.getString(cursor.getColumnIndex
(MediaStore.Files.FileColumns.DATA))
                //从 apk 文件中解析得到该安装包的程序信息
                val pi = packageManager.getPackageArchiveInfo(path,
PackageManager.GET_ACTIVITIES)
                if (pi != null) {
                    //找到包名与版本号都符合条件的 apk 文件
                    if (vc.package_name==pi.packageName &&
vc.version_name==pi.versionName) {
                        local_path = path
                    }
                }
            }
            cursor.close()
            return local_path
        }
```

5. 关于下载 APK 文件的操作过程

如果手机上没找到符合条件的安装包，就必须联网去服务器下载最新的 APK 文件。相关文件下载的 Kotlin 处理代码如下所示：

```
    //开始执行升级处理。如果本地已有安装包，就直接进行操作；如果不存在，就从网络下载安装包
    private fun startInstallApp(vc: VersionCheck) {
        appVc = vc
        //本地路径非空，表示存储卡找到最新版本的安装包，此时无须下载即可进行安装操作
        if (vc.local_path.isNotEmpty()) {
            handler.postDelayed(mInstall, 100)
        } else {
            //构建安装包下载地址的请求任务
            val down = Request(Uri.parse(vc.download_url))
            down.setAllowedNetworkTypes(Request.NETWORK_MOBILE or
Request.NETWORK_WIFI)
            //隐藏通知栏上的下载消息
            down.setNotificationVisibility(Request.VISIBILITY_HIDDEN)
            down.setVisibleInDownloadsUi(false)
            //指定下载文件的保存路径
            down.setDestinationInExternalFilesDir(this,
                    Environment.DIRECTORY_DOWNLOADS, "${vc.package_name}.apk")
            //将请求任务添加到下载队列中
            downloadId = downloader.enqueue(down)
            handler.postDelayed(mRefresh, 100)
            //弹出进度对话框，用于展示下载进度
            val message = "正在下载${appVc.app_name}的安装包"
            dialog = progressDialog(message, "请稍候")
            dialog.show()
        }
    }
```

10.6　小　　结

本章主要介绍了 Kotlin 如何进行网络通信的编程实现，包括多线程相关技术的运用（线程与消息机制、进度对话框的两种形式、异步任务的新型写法）、HTTP 接口的访问操作（JSON 串的手工解析与自动解析、HTTP 接口调用的简单实现、获取 HTTP 图片的简单实现）、文件下载的相关处理（下载管理器的用法、自定义文本进度圈、在页面上动态显示下载进度）以及 Content 内容组件的数据存储和读取（内容提供器、内容解析器、内容观察器）。最后设计了一个实战项目“电商 App 的自动升级”，在该项目的 Kotlin 编码中采用了前面介绍的部分网络通信技术，以及通过内容组件自动判断是否存在已下载的安装包，另外还介绍了 Kotlin 对可变字符串的改进编码。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会使用 Kotlin 实现多线程任务的开发，重点掌握 Kotlin 对线程、进度对话框、异步任务的简要写法。

（2）学会使用 Kotlin 完成 HTTP 接口的访问编码，重点掌握 Kotlin 如何自动解析 JSON 串、如何便捷调用 HTTP 接口、如何快速获取 HTTP 图片。

（3）学会使用 Kotlin 进行文件下载的相关操作，重点掌握 Kotlin 对下载事件的处理，以及 Kotlin 是怎样轮询下载进度的。

（4）学会使用 Kotlin 开发 Content 内容组件的功能，例如封装数据的对外接口，以及对开放内容接口的系统数据进行查询、修改和监视操作等。

（5）学会利用 Kotlin 简化可变字符串的编码过程。