电量优化

电池续航时间是移动用户体验中最重要的一个方面。没电的设备完全无法使用。因此，对于应用来说，尽可能地考虑电池续航时间是至关重要的。在我们开发时对于单个APP应该注意能够：

- **减少操作**：您的应用是否存在可删减的多余操作？例如，是否可以缓存已下载的数据，而不是每次重新下载数据？
- **推迟操作**：应用是否需要立即执行某项操作？例如，是否可以等到设备充电后或者Wifi连接时（通常情况下使用移动网络要比WIFI更耗电 ）再将数据备份到云端？
- **合并操作**：工作是否可以批处理，而不是多次将设备置于活动状态？比如和服务器请求不同的接口获取数据，部分接口是否可以合并为一个？

Doze低电耗模式和StandBy待机模式

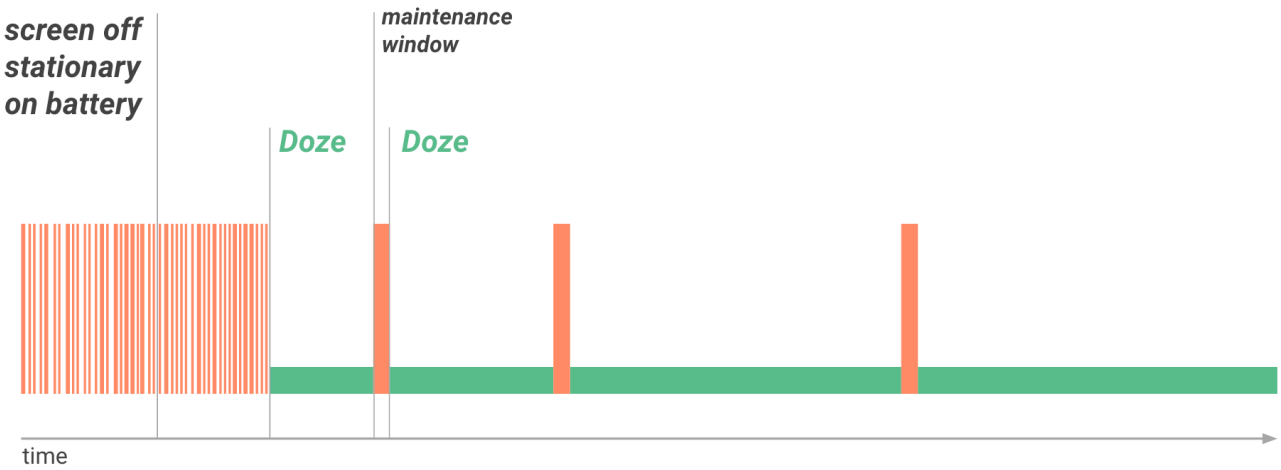
从 Android 6.0（API 级别 23）开始，Android 引入了两项省电功能，通过管理应用在设备未连接至电源时的行为方式，帮助用户延长电池寿命。当用户长时间未使用设备时，**低电耗模式会延迟应用的后台 CPU 和网络活动**，从而降低耗电量。**应用待机模式会延迟用户近期未与之交互的应用的后台网络活动。**

低电耗模式和应用待机模式管理在 Android 6.0 或更高版本上运行的所有应用的行为，无论它们是否专用于 API 级别 23。

Doze低电耗模式

如果设备 **未充电、屏幕熄灭、让设备在一段时间内保持不活动状态** ，那么设备就会进入Doze模式。在Doze模式下，系统会尝试通过限制应用访问占用大量网络和 CPU 资源的服务来节省电量。它还会阻止应用访问网络，并延迟其作业、同步和标准闹钟。

Doze中文是打盹，所以系统会定期退出打盹一小段时间，让应用完成其延迟的活动。在此维护期内，系统会运行所有待处理的同步、作业和闹钟，并允许应用访问网络。



随着时间的推移，系统安排维护期的次数越来越少，这有助于在设备未连接至充电器的情况下长期处于不活动状态时降低耗电量。

一旦用户通过移动设备、打开屏幕或连接至充电器唤醒设备，系统就会立即退出低电耗模式，并且所有应用都会恢复正常活动。

在低电耗模式下，您的应用会受到以下限制：

- 暂停访问网络。
- 系统忽略*PowerManager.WakeLock*唤醒锁定。
- 标准 `AlarmManager` 闹钟（包括 `setExact()` 和 `setWindow()`）推迟到下一个维护期。
  - 如果需要设置在设备处于低电耗模式时触发的闹钟，请使用API 23(6.0)提供的 `setAndAllowWhileIdle()` (一次性闹钟，同set方法)或 `setExactAndAllowWhileIdle()` （比set方法设置的精度更高，同setExact）。
  - 使用 `setAlarmClock()` 设置的闹钟将继续正常触发，系统会在这些闹钟触发之前不久退出低电耗模式。
- 系统不执行 WLAN 扫描。
- 系统不允许运行同步适配器*AbstractThreadedSyncAdapter* (账号同步拉活)。
- 系统不允许运行 `JobScheduler`。

> `setAndAllowWhileIdle()` 及 `setExactAndAllowWhileIdle()` 为每个应用触发闹钟的频率都不能超过每 9 分钟一次。
>
> 如果应用需要与网络建立持久性连接来接收消息，应尽可能使用 Firebase 云消息传递 (FCM)。如果是国内无法使用Google服务，需要实现如IM功能，需要与手机厂商合作。

Standby待机模式

应用待机模式允许系统判定应用在用户未主动使用它时是否处于待机状态。当用户有一段时间未触摸应用并且应用没有以下表现，则Android系统就会使应用进入空闲状态

- 应用当前有一个进程在前台运行（作为活动或前台服务，或者正在由其他活动或前台服务使用）。
- 应用生成用户可在锁定屏幕或通知栏中看到的通知。

当用户将设备插入电源时，系统会从待机状态释放应用，允许它们自由访问网络并执行任何待处理的作业和同步。如果设备长时间处于闲置状态，系统将允许闲置应用访问网络，频率大约每天一次。

白名单

系统提供了一个可配置的白名单，将部分免除低电耗模式和应用待机模式优化的应用列入其中。 在低电耗模式和应用待机模式期间，列入白名单的应用可以使用网络并保留部分唤醒锁定。不过，列入白名单的应用**仍会受到其他限制**，就像其他应用一样。例如，列入白名单的应用的作业和同步会延迟（在6.0及以下的设备上），并且其常规 `AlarmManager` 闹钟不会触发。应用可以调用 `PowerManager.isIgnoringBatteryOptimizations()` 来检查应用当前是否在豁免白名单中。

可以在**设置 中的 电池优化**手动配置白名单 。另外，系统也提供了一些方法，让应用要求用户将其列入白名单。

- 应用可以触发 `ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS` Intent，让用户直接转到**电池优化**，以便他们在其中添加该应用。

```
startActivity(new Intent(Settings.ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS));
```

- 具有 `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` 权限的应用可以触发一个系统对话框，让用户直接将该应用添加到白名单，而无需转到“设置”。此类应用将通过触发 `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` Intent 来触发该对话框。

```
Intent intent = new Intent(Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS);
intent.setData(Uri.parse("package:"+getPackageName()));
startActivity(intent);

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

在低电耗模式下测试

```
#启用Doze
adb shell dumpsys deviceidle enable
#强制进入doze模式（同时还需要关闭屏幕）
adb shell dumpsys deviceidle force-idle

#退出doze模式
adb shell dumpsys deviceidle unforce
#关闭doze
adb shell dumpsys deviceidle disable
#重置设备
adb shell dumpsys battery reset


#查看doze白名单
adb shell dumpsys deviceidle whitelist
```

在应用待机模式下测试

```
#设置断开充电
adb shell dumpsys battery unplug
#进入standby
adb shell am set-inactive <packageName> true

#退出standby
adb shell am set-inactive <packageName> false
#查看是否处于standby
adb shell am get-inactive <packageName>
#重置
adb shell dumpsys battery reset
```

监控电池电量和充电状态

为了减少电池续航被我们软件的影响，我们可以通过检查电池状态以及电量来判断是否进行某些操作。比如我们可以在充电时才进行一些数据上报之类的操作。

获取充电状态

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = registerReceiver(null, ifilter);

// 是否正在充电
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                status == BatteryManager.BATTERY_STATUS_FULL;

// 什么方式充电?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
//usb
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
//充电器
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;

Log.e(TAG, "isCharging: " + isCharging + " usbCharge: " + usbCharge + " acCharge:" + acCharge);
```

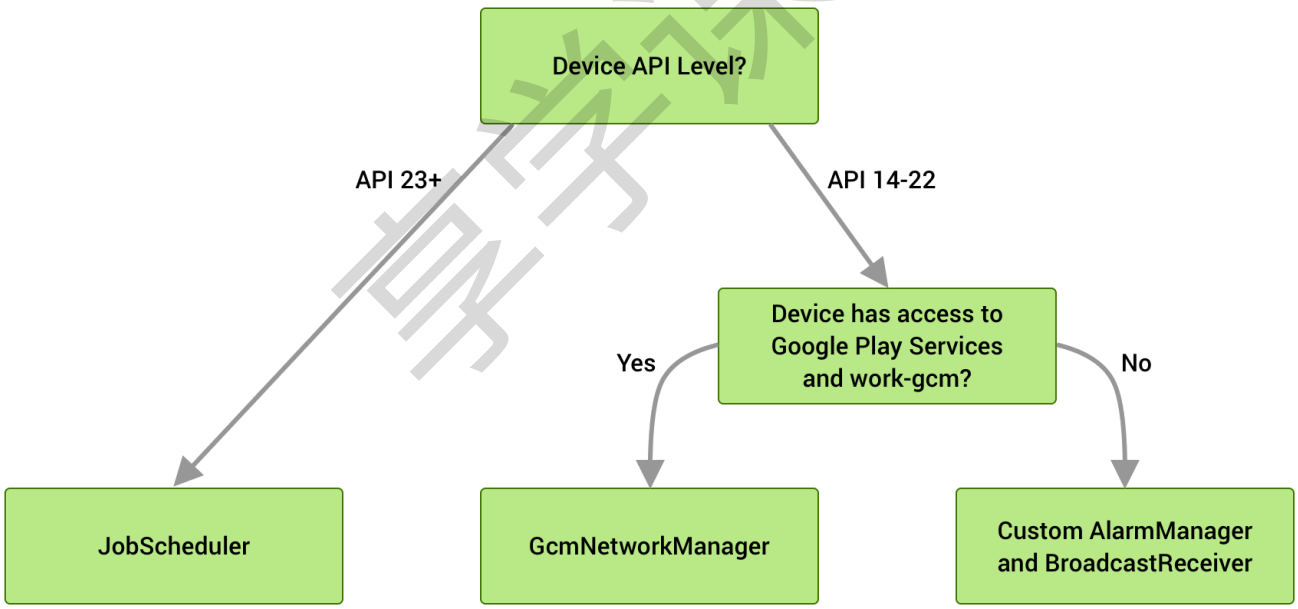
监控充电状态变化

```
//注册广播
IntentFilter ifilter = new IntentFilter();
//充电状态
ifilter.addAction(Intent.ACTION_POWER_CONNECTED);
ifilter.addAction(Intent.ACTION_POWER_DISCONNECTED);
//电量显著变化
ifilter.addAction(Intent.ACTION_BATTERY_LOW); //电量不足
```

```
ifilter.addAction(Intent.ACTION_BATTERY_OKAY); //电量从低变回高
powerConnectionReceiver = new PowerConnectionReceiver();
registerReceiver(powerConnectionReceiver, ifilter);

public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(Intent.ACTION_POWER_CONNECTED)) {
            Toast.makeText(context, "充电状态：CONNECTED", Toast.LENGTH_SHORT).show();
        } else if (intent.getAction().equals(Intent.ACTION_POWER_DISCONNECTED)) {
            Toast.makeText(context, "充电状态：DISCONNECTED", Toast.LENGTH_SHORT).show();
        } else if (intent.getAction().equals(Intent.ACTION_BATTERY_LOW)) {
            Toast.makeText(context, "电量过低", Toast.LENGTH_SHORT).show();
        } else if (intent.getAction().equals(Intent.ACTION_BATTERY_OKAY)) {
            Toast.makeText(context, "电量从低变回高", Toast.LENGTH_SHORT).show();
        }
    }
}
```

WorkManager

WorkManager API 是一个针对原有的 Android 后台调度 API 整合的建议替换组件。



如果设备在 API 级别 23 或更高级别上运行，系统会使用 JobScheduler。在 API 级别 14-22 上，系统会使用 GcmNetworkManager（如果可用），否则会使用自定义 AlarmManager 和 BroadcastReciever 实现作为备用。

```
Constraints constraints = new Constraints.Builder()
                .setRequiredNetworkType(NetworkType.UNMETERED)  //Wi-Fi
                .setRequiresCharging(true) //在设备充电时运行
                .setRequiresBatteryNotLow(true) //电量不足不会运行
                .build();

 OneTimeWorkRequest uploadWorkRequest =
                new OneTimeWorkRequest.Builder(UploadWorker.class)
                        .setConstraints(constraints)
                        .build();

 WorkManager.getInstance(this)
            .enqueueUniqueWork("upload", ExistingWorkPolicy.KEEP,uploadWorkRequest);
```

Battery Historian

Battery Historian是一个可以了解设备随时间的耗电情况的工具 。在系统级别，该工具以 HTML 的形式可视化来自系统日志的电源相关事件。在具体应用级别，该工具可提供各种数据，帮助您识别耗电的应用行为。

Battery Historian可以帮助我们查看应用是否具有以下耗电行为：

- 过于频繁地触发唤醒提醒（至少每 10 秒钟一次）。
- 持续保留 GPS 锁定。
- 至少每 30 秒调度一次作业。
- 至少每 30 秒调度一次同步。
- 使用手机无线装置的频率高于预期。

https://github.com/google/battery-historian

如何安装在github主页有详细步骤，也可以查看Zero老师的博客：https://www.jianshu.com/p/378cf678bdeb

收集数据

1、重置电池数据收集

- `adb shell dumpsys batterystats --reset`

2、开启wakeLock唤醒锁信息记录（可选）

- `adb shell dumpsys batterystats --enable full-wake-history`

2、断开设备与计算机的连接，以便仅消耗设备电池的电量。

3、使用您的应用并执行您想要获取数据的操作；例如，断开 WLAN 连接并将数据发送到云端。

4、重新连接手机。

5、生成报告

- 对于搭载 Android 7.0 及更高版本的设备：

  ```
  adb bugreport > [path/]bugreport.zip
  ```

- 对于搭载 Android 6.0 及更低版本的设备：

  ```
  adb bugreport > [path/]bugreport.txt
  ```

关闭wakeLock统计：

```
adb shell dumpsys batterystats --disable full-wake-history
```

报告分析

使用系统级视图

Battery Historian 工具提供各种应用和系统行为的系统级可视化结果，以及它们随时间推移与耗电量的相关性。如图 1 所示，此视图可帮助您诊断和识别应用存在的耗电问题。

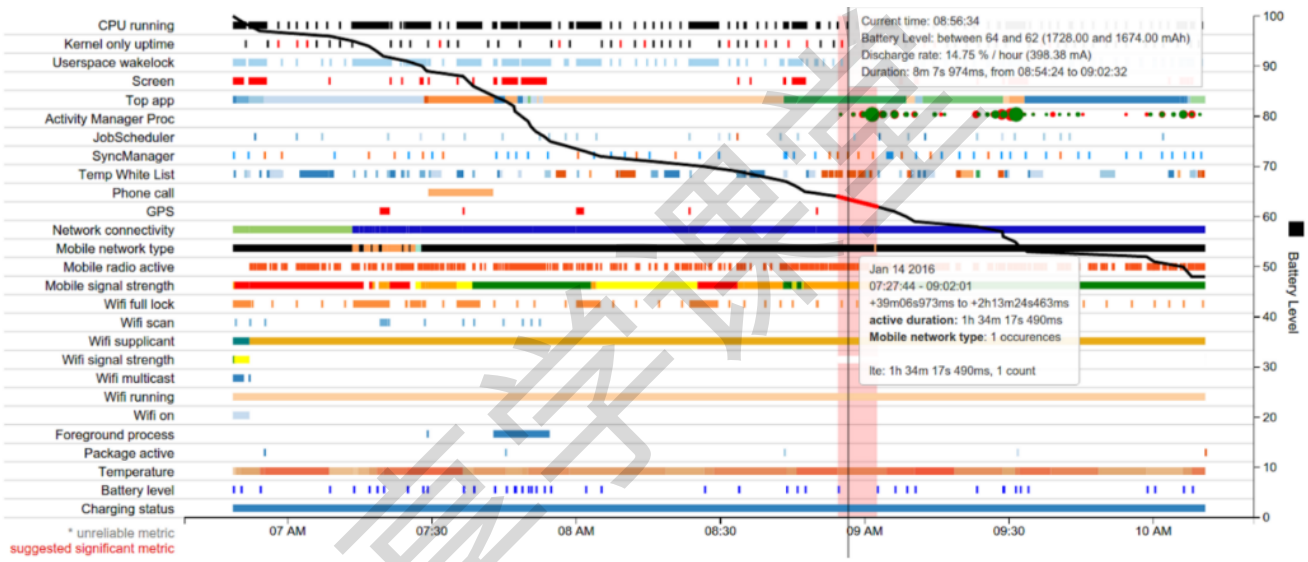


图 1. Battery Historian 显示了影响功耗的系统级事件。

该图中特别值得注意的是表示电池电量的黑色水平下降趋势线（在 y 轴上进行测量）。例如，在电池电量线的最开始处，大约早上 6:50 时，该图表显示电量出现较为急剧的下降。

图 2 为该部分显示画面的特写图。

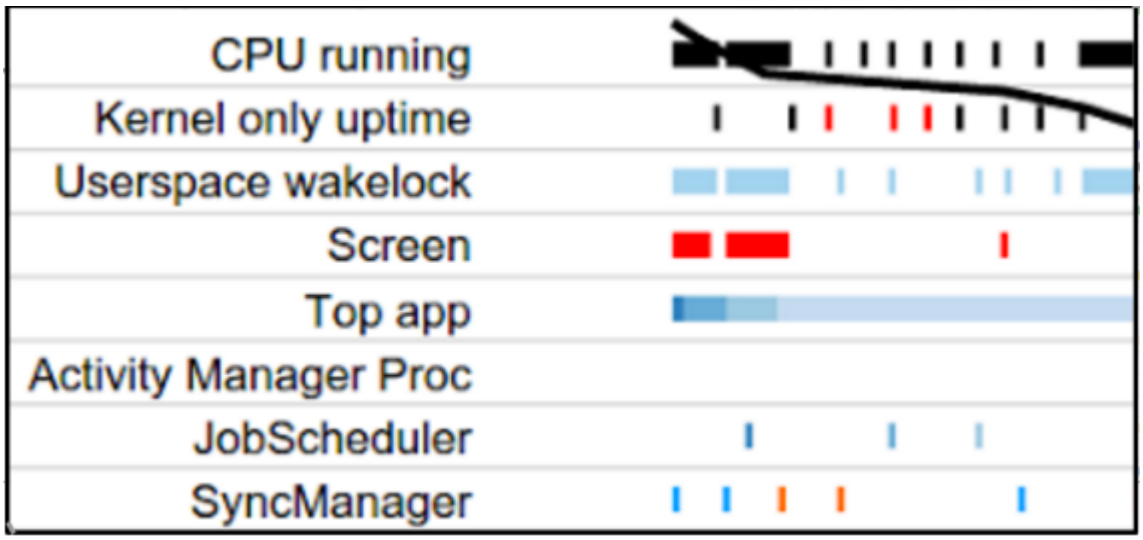


图 2. Battery Historian 时间轴（大约从早上 6:50 到 7:20）的特写图。

在电池电量线的最开始处，随着电池电量的急剧下降，显示画面上显示有三件事正在发生：CPU 正在运行，应用已获取唤醒锁定，且屏幕已打开。通过这种方式，Battery Historian 可帮助您了解耗电量高时正在发生什么事件。然后，您可以针对应用中的这些行为，研究是否可以进行相关优化。

查看具体应用的数据

表格提供了关于您的应用的两个数据维度。首先，您可以查找应用的耗电量与其他应用相比的排名位置。为此，请点击“Tables”下的“Device Power Estimates”表格。

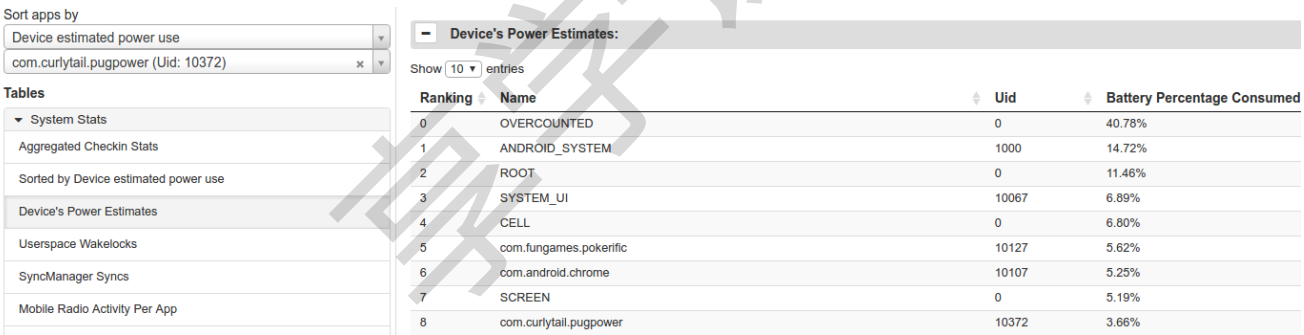


图 3. 研究哪些应用的耗电量最大。

图 3 中的表格显示，Pug Power 在此设备上的耗电量排名第九，在不属于操作系统的应用中排名第三。这些数据表明此应用需要更深入的研究。

要查找特定应用的数据，在可视化图表左侧下方，在“App Selection”下方的第二个下拉菜单中输入应用的软件包名称。

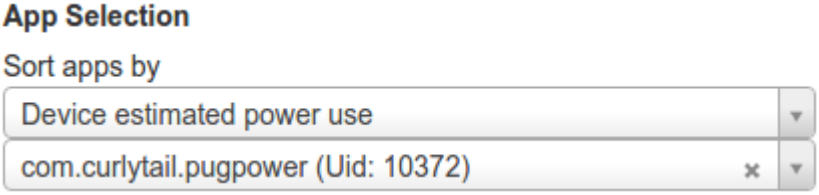


图 4. 输入特定的应用以查看其数据。

图 5 和图 6 显示了 PugPower 的数据：图 5 显示了具体应用数据的可视化图表，图 6 显示了相应的表格数据。

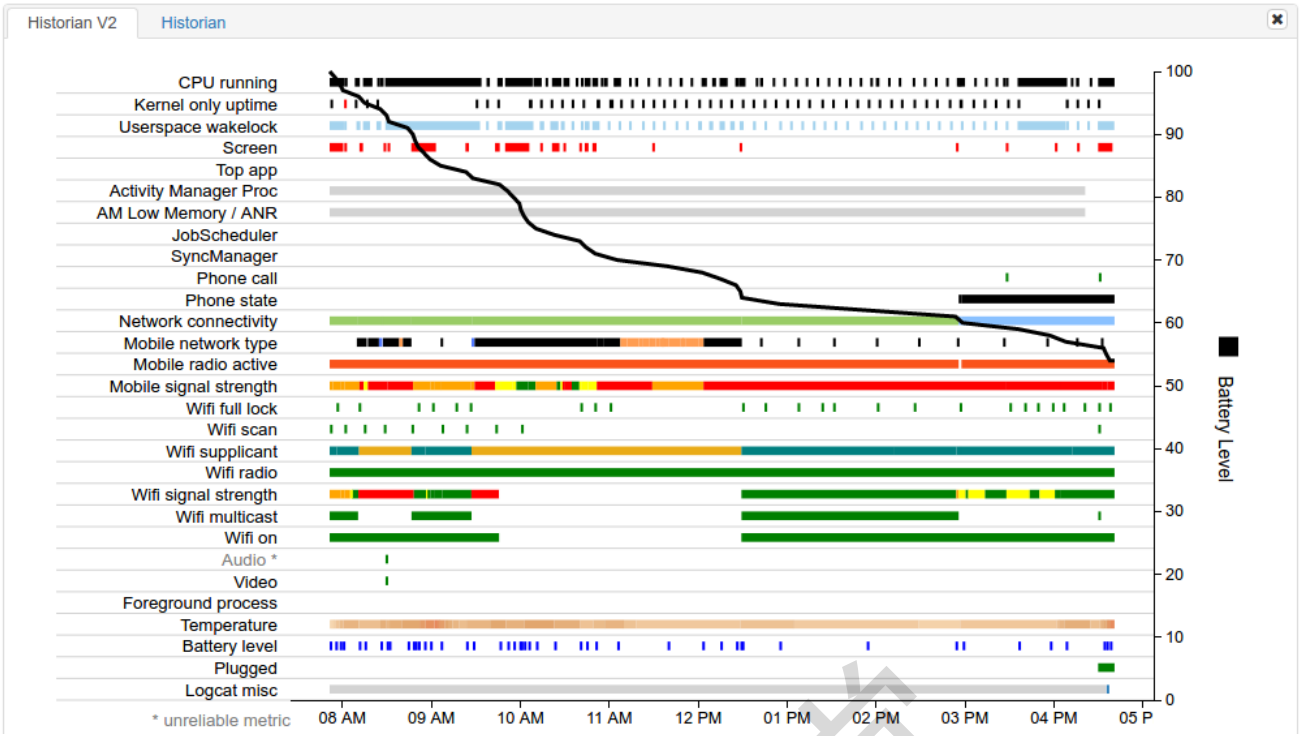


图 5. 虚构应用 Pug Power 的数据可视化图表。

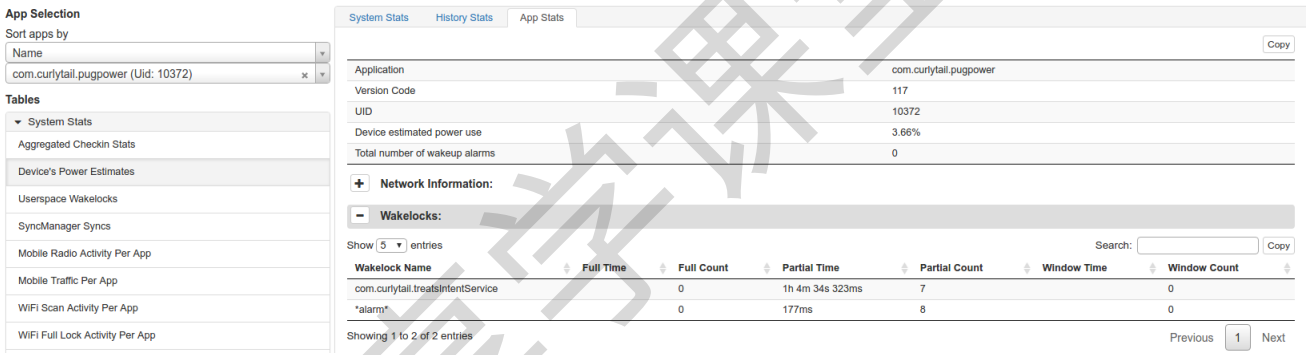


图 6. 虚构应用 Pugle Power 的表格数据。

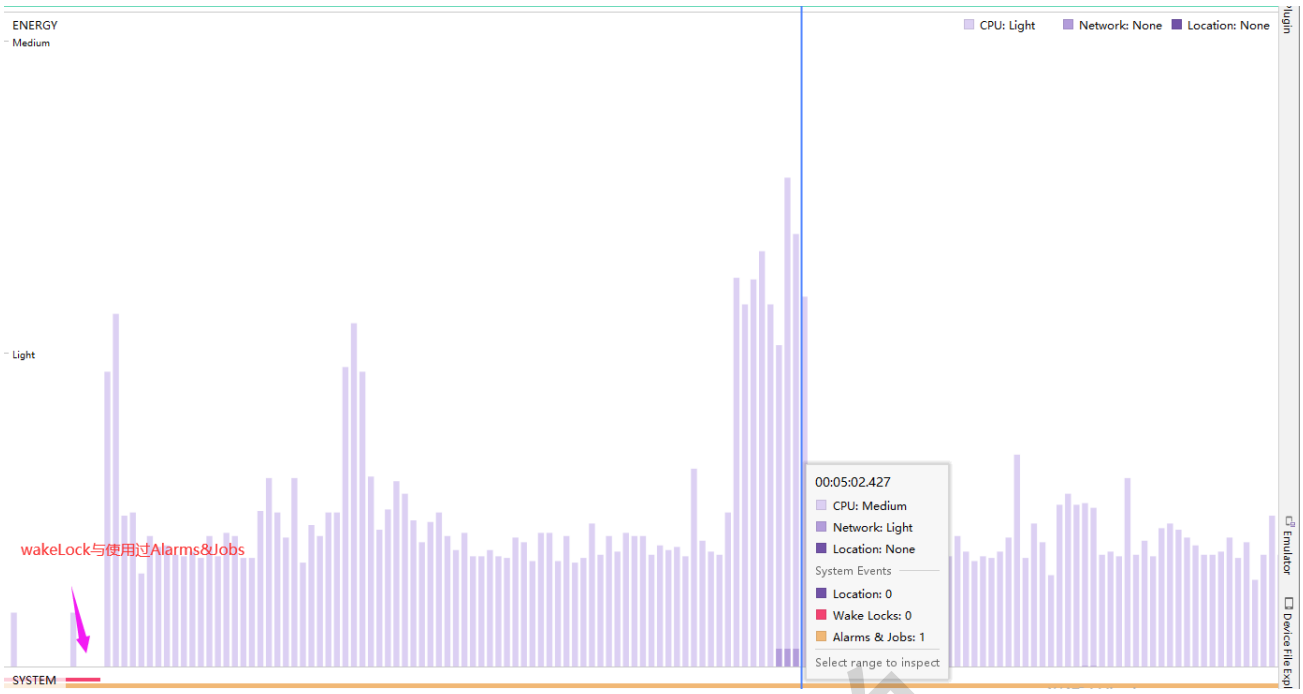
查看可视化图表并没有发现任何显而易见的问题。JobScheduler 行显示应用没有调度作业。SyncManager 行显示应用尚未执行任何同步操作。

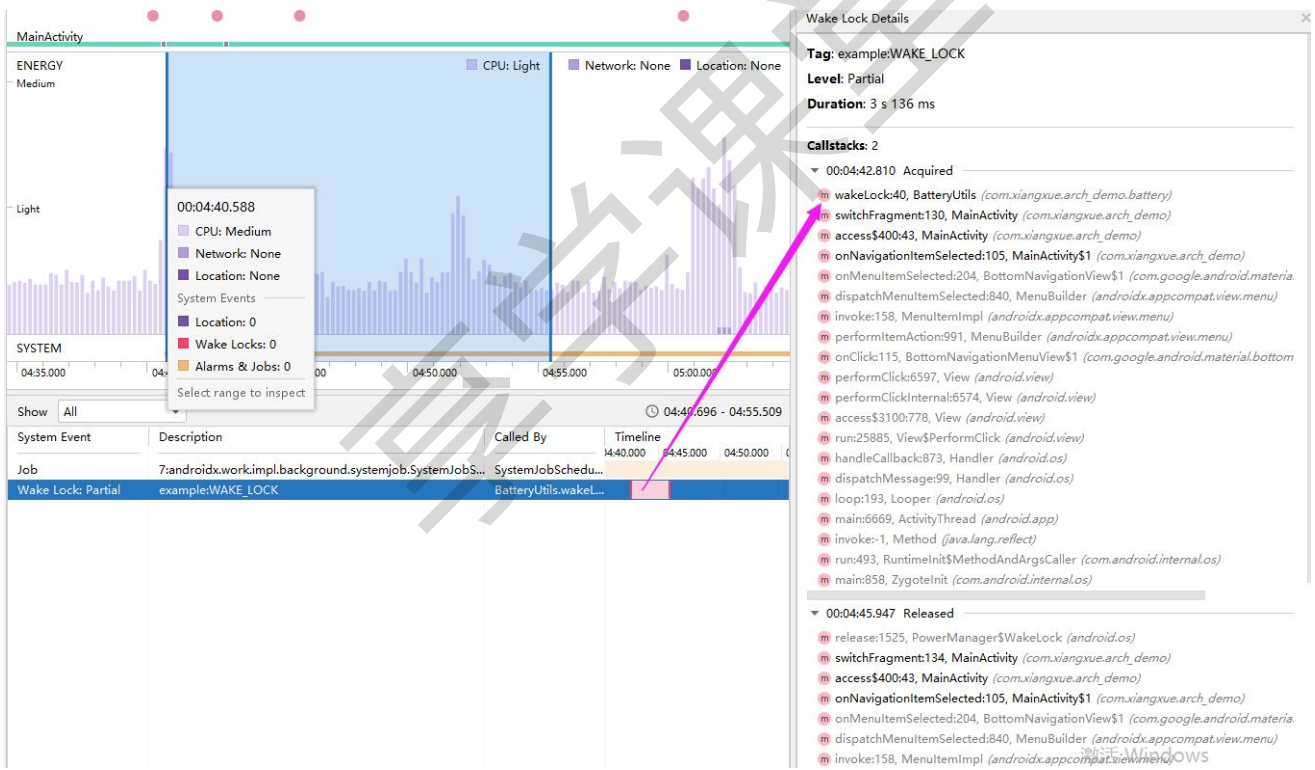
不过，查看表格数据的“唤醒锁定”部分发现，Pug Power 获取了 1 个小时内的唤醒锁定总次数。这种代价高昂的异常行为可能是应用耗电量较高的原因。这条信息有助于开发者专攻优化后可能会大大获益的方面。在这种情况下，我们需要结合代码考虑为什么应用会获得如此多的唤醒锁定时间？是否可以优化？

Energy Profiler

使用 Android 8.0 及以上版本的设备时，使用Energy Profiler 可以了解应用在哪里耗用了不必要的电量。 Energy Profiler 会监控 CPU、网络无线装置和 GPS 传感器的使用情况，并直观地显示其中每个组件消耗的电量。还会显示可能会影响耗电量的系统事件（唤醒锁定、闹钟、作业和位置信息请求）的发生次数。

使用Profile 运行App。





网络优化

正常一条网络请求需要经过的流程是这样：

- DNS 解析，请求DNS服务器，获取域名对应的 IP 地址；
- 与服务端建立连接，包括 tcp 三次握手，安全协议同步流程；
- 连接建立完成，发送和接收数据，解码数据。

这里有明显的三个优化点：

- 直接使用 IP 地址，去除 DNS 解析步骤；
- 不要每次请求都重新建立连接，复用连接或一直使用同一条连接(长连接)；
- 压缩数据，减小传输的数据大小。

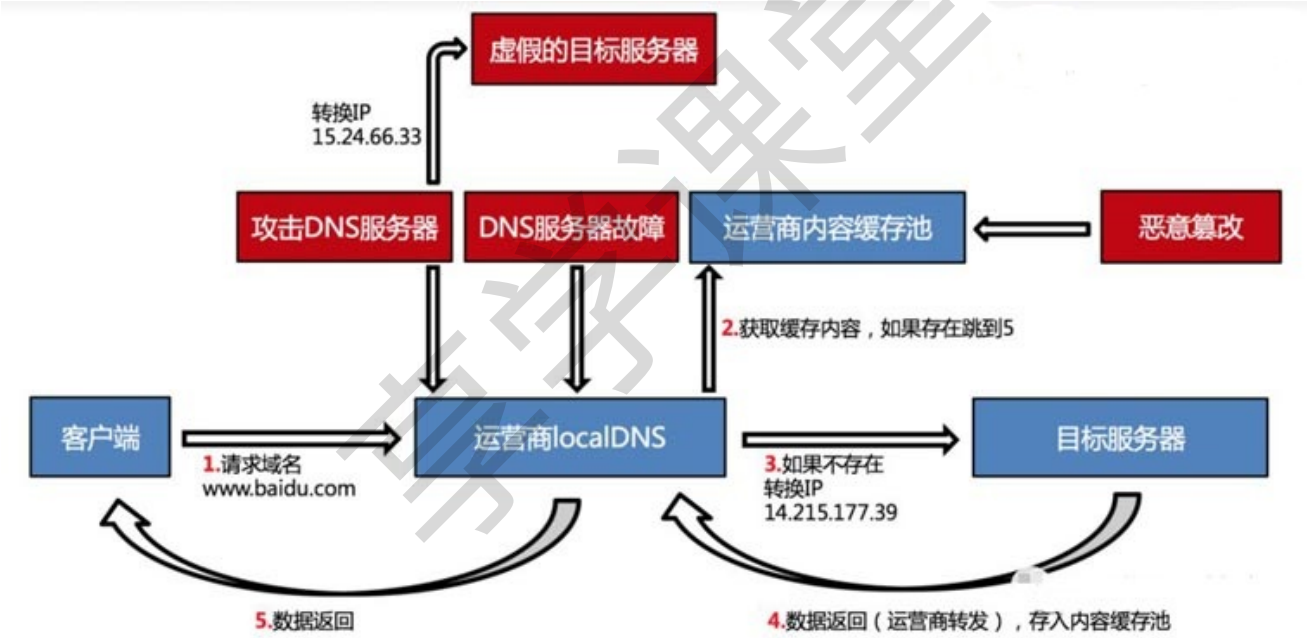
DNS优化

DNS（Domain Name System），它的作用是根据域名查出IP地址，它是HTTP协议的前提，只有将域名正确的解析成IP地址后，后面的HTTP流程才能进行。

DNS 完整的解析流程很长，会先从本地系统缓存取，若没有就到最近的 DNS 服务器取，若没有再到主域名服务器取，每一层都有缓存，但为了域名解析的实时性，每一层缓存都有过期时间。

传统的DNS解析机制有几个缺点：

- 缓存时间设置得长，域名更新不及时，设置得短，大量 DNS 解析请求影响请求速度；
- 域名劫持，容易被中间人攻击，或被运营商劫持，把域名解析到第三方 IP 地址，据统计劫持率会达到7%；
- DNS 解析过程不受控制，无法保证解析到最快的IP；
- 一次请求只能解析一个域名。



为了解决这些问题，就有了 HTTPDNS，原理很简单，就是自己做域名解析的工作，通过 HTTP 请求后台去拿到域名对应的 IP 地址，直接解决上述所有问题。

HTTPDNS的好处总结就是：

- Local DNS 劫持：由于 HttpDns 是通过 IP 直接请求 HTTP 获取服务器 A 记录地址，不存在向本地运营商询问 domain 解析过程，所以从根本避免了劫持问题。
- DNS 解析由自己控制，可以确保根据用户所在地返回就近的 IP 地址，或根据客户端测速结果使用速度最快的 IP；
- 一次请求可以解析多个域名。
- ……

HTTPDNS 几乎成为中大型 APP 的标配。解决了第一个问题， DNS 解析耗时的问题，顺便把DNS 劫持也解决了。

> 阿里HttpDNS：https://help.aliyun.com/document_detail/30103.html?spm=a2c4g.11186623.6.543.7eee78bc3kDYhO

连接优化

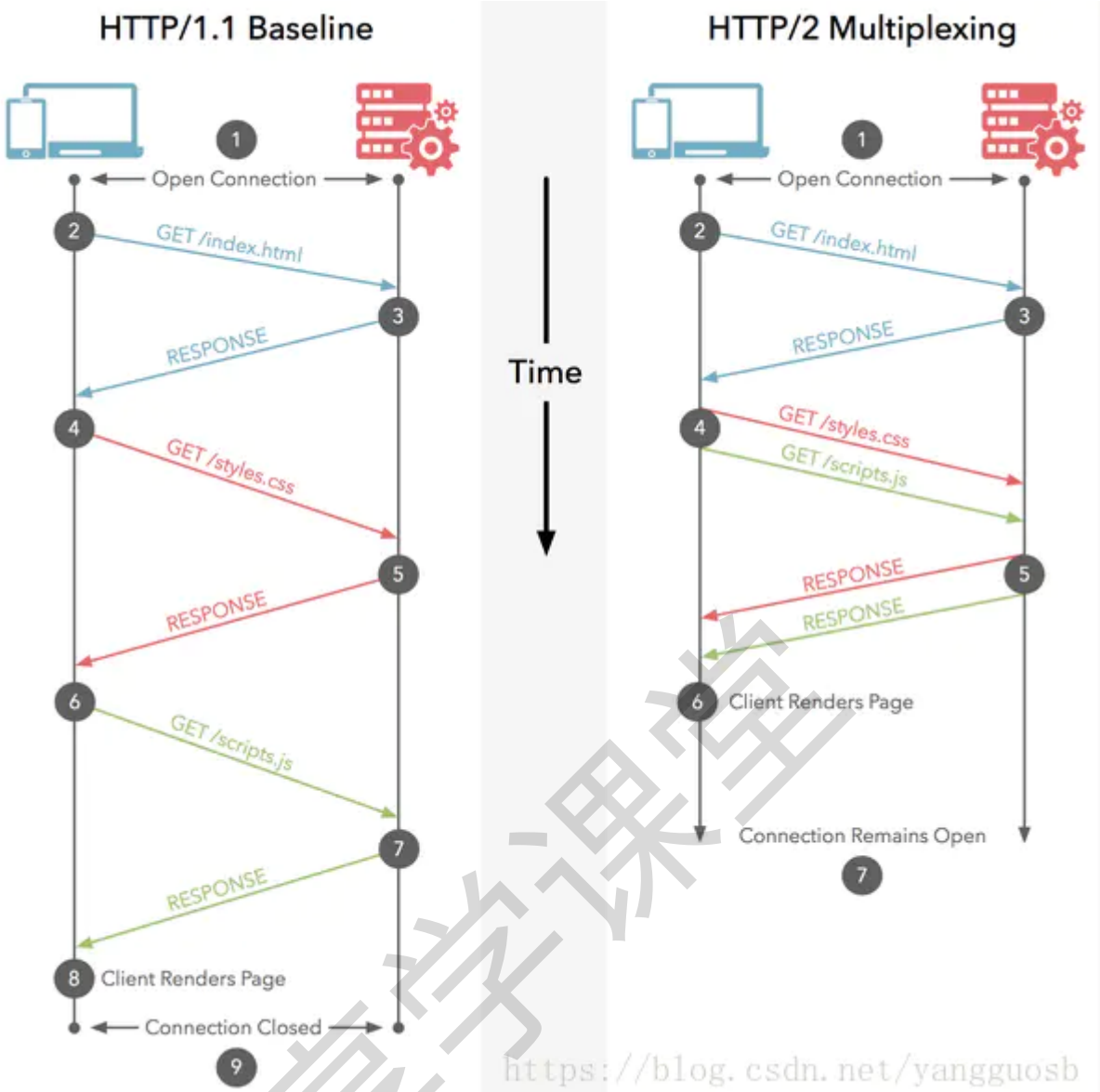
第二个问题，连接建立耗时的问题，这里主要的优化思路是复用连接，不用每次请求都重新建立连接，如何更有效率地复用连接，可以说是网络请求速度优化里最主要的点了。

【keep-alive】： HTTP 协议里有个 keep-alive，HTTP1.1默认开启，一定程度上缓解了每次请求都要进行TCP三次握手建立连接的耗时。原理是请求完成后不立即释放连接，而是放入连接池中，若这时有另一个请求要发出，请求的域名和端口是一样的，就直接拿出连接池中的连接进行发送和接收数据，少了建立连接的耗时。 实际上现在无论是客户端还是浏览器都默认开启了keep-alive，对同个域名不会再有每发一个请求就进行一次建连的情况，纯短连接已经不存在了。

但有 `keep-alive` 的连接一次只能发送接收一个请求，在上一个请求处理完成之前，无法接受新的请求。若同时发起多个请求，就有两种情况：

- 若串行发送请求，可以一直复用一个连接，但速度很慢，每个请求都要等待上个请求完成再进行发送。
- 若并行发送请求，那么只能每个请求都要进行tcp三次握手建立新的连接。

对并行请求的问题，新一代协议 HTTP2 提出了多路复用去解决。 HTTP2 的多路复用机制一样是复用连接，但它复用的这条连接支持同时处理多条请求，所有请求都可以并发在这条连接上进行，也就解决了上面说的并发请求需要建立多条连接带来的问题。



多路复用把在连接里传输的数据都封装成一个个stream，每个stream都有标识，stream的发送和接收可以是乱序的，不依赖顺序，也就不会有阻塞的问题，接收端可以根据stream的标识去区分属于哪个请求，再进行数据拼接，得到最终数据。

Android 的开源网络库**OKhttp**默认就会开启`keep-alive`，并且在**Okhttp3**以上版本也支持了 HTTP2。

数据压缩

第三个问题，传输数据大小的问题。数据对请求速度的影响分两方面，一是压缩率，二是解压序列化反序列化的速度。目前最流行的两种数据格式是 json 和 protobuf，json 是字符串，protobuf 是二进制，即使用各种压缩算法压缩后，protobuf 仍会比 json 小，数据量上 protobuf 有优势，序列化速度 protobuf 也有一些优势 。

https://github.com/protocolbuffers/protobuf/blob/master/java/lite.md

除了选择不同的序列化方式（数据格式）之外，Http可以对内容（也就是body部分）进行编码，可以采用gzip这样的编码，从而达到压缩的目的。

在**OKhttp**的`BridgeInterceptor`中会自动为我们开启gzip解压的支持。

```
boolean transparentGzip = false;
if (userRequest.header("Accept-Encoding") == null && userRequest.header("Range") == null) {
      transparentGzip = true;
      requestBuilder.header("Accept-Encoding", "gzip");
}
```

如果服务器响应头存在：`Content-Encodin:gzip`

```
//服务器响应 Content-Encodin:gzip 并且有响应body数据
if (transparentGzip
    && "gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))
    && HttpHeaders.hasBody(networkResponse)) {
  GzipSource responseBody = new GzipSource(networkResponse.body().source());
  Headers strippedHeaders = networkResponse.headers().newBuilder()
      .removeAll("Content-Encoding")
      .removeAll("Content-Length")
      .build();
  responseBuilder.headers(strippedHeaders);
  String contentType = networkResponse.header("Content-Type");
  responseBuilder.body(new RealResponseBody(contentType, -1L, Okio.buffer(responseBody)));
}
```

客户端也可以发送压缩数据给服务端，通过代码将请求数据压缩，并在请求中加入`Content-Encodin:gzip`即可。

```
private RequestBody gzip(final RequestBody body) {
        return new RequestBody() {
            @Override
            public MediaType contentType() {
                return body.contentType();
            }

            @Override
            public long contentLength() {
                return -1; // We don't know the compressed length in advance!
            }

            @Override
            public void writeTo(BufferedSink sink) throws IOException {
                BufferedSink gzipSink = Okio.buffer(new GzipSink(sink));
                body.writeTo(gzipSink);
                gzipSink.close();
            }
};

public RequestBody getGzipRequest(String body) {
        RequestBody request = null;
        try {
            request = RequestBody.create(
                MediaType.parse("application/octet-stream"),compress(body)
            );
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        return request;
}
```

其他

1、使用webp代替png/jpg

2、不同网络的不同图片下发，如（对于原图是300x300的图片）：

- 2/3G使用低清晰度图片：使用100X100的图片;
- 4G再判断信号强度为强则使用使用300X300的图片，为中等则使用200x200，信号弱则使用100x100图片;
- WiFi网络：直接下发300X300的图片

3、http开启缓存 / 首页数据加入缓存

附录一： Ptotobuf使用方式

Protobuf是一种平台无关、语言无关、可扩展且轻便高效的**序列化**数据结构的协议，可以用于**网络通信**和**数据存储**。 可简单类比于 XML ，其具有以下特点：

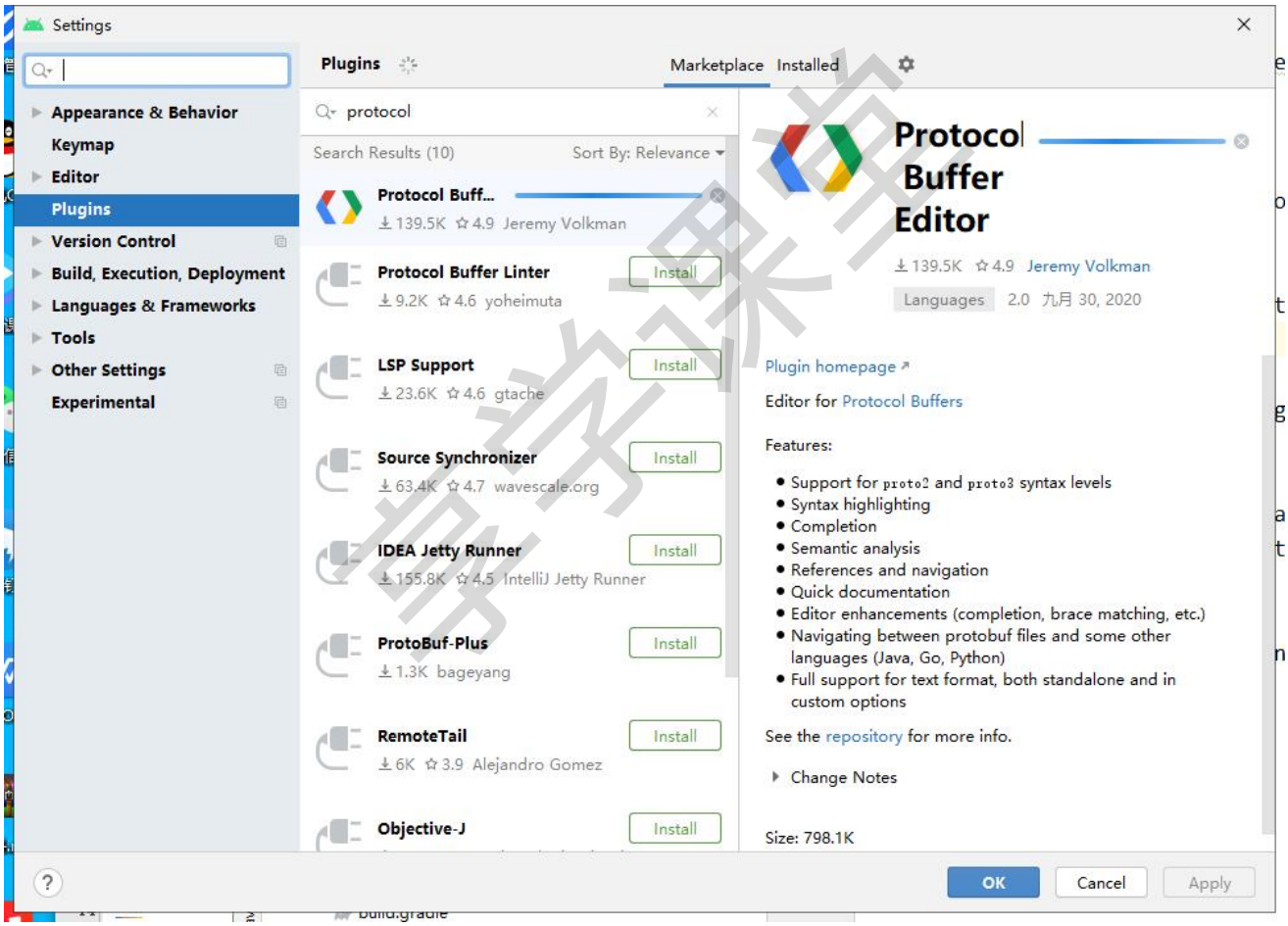
- **语言无关、平台无关**。即 ProtoBuf 支持 Java、C++、Python 等多种语言，支持多个平台
- **高效**。即比 XML 更小（3 ~ 10倍）、更快（20 ~ 100倍）、更为简单
- **扩展性、兼容性好**。你可以更新数据结构，而不影响和破坏原有的旧程序

一、引入插件

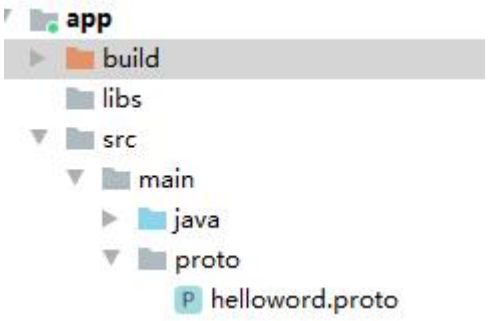
```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.13'
  }
}

apply plugin: 'com.google.protobuf'

protobuf {
    protoc {
        artifact = 'com.google.protobuf:protoc:3.11.0'
    }
```

```
    generateProtoTasks {
        all().each { task ->
            task.builtins {
                java {
                    option "lite"
                }
            }
        }
    }
}

dependencies{
     implementation 'com.google.protobuf:protobuf-javalite:3.11.0'
}
```

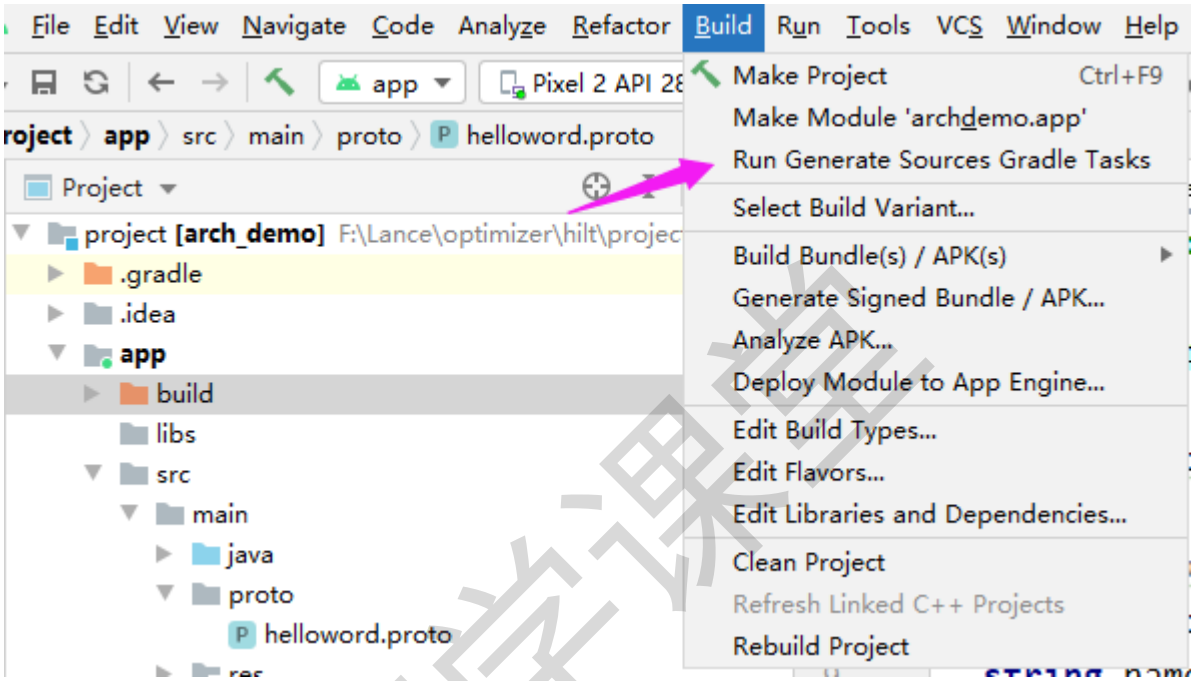
二、安装IDEA插件并编写proto文件



在Java同级目录下创建*proto*目录



三、执行编译



四、开始使用

```
Helloword.HelloRequest helloRequest =
                Helloword.HelloRequest.newBuilder().setName("Lance")
                        .setAge(18)
                        .build();
//获得序列化后数据大小
helloRequest.getSerializedSize();

//序列化
byte[] bytes = helloRequest.toByteArray();
//反序列化
try {
    Helloword.HelloRequest helloRequest1 = Helloword.HelloRequest.parseFrom(bytes);
} catch (InvalidProtocolBufferException e) {
    e.printStackTrace();
}
```

混淆配置：

```
-keep class * extends com.google.protobuf.GeneratedMessageLite { *; }
```

附录二：Protobuf数据结构

Proto	C++	Java	Python
double	double	double	Float
float	float	float	Float
int32	int32	int	int
int64	int64	long	int/long
uint32	uint32	int	int/long
uint64	uint64	long	int/long
bool	bool	boolean	bool
string	string	String	
bytes	string	ByteString	str