内核编码规范

版本说明

版本号	作者	更新日期
1.0.0	蔡小波	2018.05.15
1.0.1	蔡小波	2018.06.05
1.0.2	蔡小波	2018.06.25
1.0.3	蔡小波	2018.07.06
1.0.4	蔡小波	2018.08.01
1.0.5	蔡小波	2018.10.17
1.0.6	蔡小波	2018.11.29
1.0.7	蔡小波	2019.01.07

前言

简介

《内核编码规范》作为软件开发重要的一环，核心目标：**通过规约，提升编码水平、软件开发质量** 子目标： 防患未然，提升质量意识，降低故障率和维护成本； 标准统一，提升效率； 追求卓越的工匠精神，打磨精品代码。

内容来源：

1. 外部（google等互联网公司）
2. 平时代码评审的共性问题

使用说明

本手册是收集行业里面发布的优秀的学习资源，以及我们平时开发过程中，总结提炼出来的共性的、易出错的问题点，简明扼要展示出来，并要求我们大家一起去遵守、执行和维护。

PS：大家平时遇到的典型的、有参考价值的问题以及解决规范可以更新到这里，由部门DR负责维护、更新 本手册最终解释权归部门所有

其他相关规范

1.公司：Android-Java规范：http://km.vivo.xyz/pages/viewpage.action?pageId=13599278

2.公司：移动安全应用规范：http://km.vivo.xyz/pages/viewpage.action?pageId=13599280

3.部门：Android编码规范：http://km.vivo.xyz/pages/viewpage.action?pageId=16270926

4.部门：vivo自定义接口方法使用规范:http://km.vivo.xyz/pages/viewpage.action?pageId=16270924

一.代码设计原则

1.1 依赖倒置原则

1.1 要求清晰chromium代码的层次关系。

1.2 上层代码可以依赖下层代码，不能反向依赖。

1.3 下层代码希望调用上层的方法时，定义client，delegate，observer，listener等，然后在上层代码实现。

1.4 模块同样存在依赖关系，也不能出现反向依赖，目前在每个模块的DEPS文件中都定义了本模块可以依赖的模块，DEPS未制定依赖的模块，代码中include头文件就会报错。

1.2 功能实现上移

在功能设计的时候，要考虑功能的主要逻辑在Chromium的各个代码层次上实现的可能性。

一般来说，越往上层，代码层次结构会越清晰，逻辑上会越简单，但代码的修改量会越大，逻辑的粒度会越大。

但从chromium的代码设计风格来看，更倾向于这种方式。

1.3 UI线程要求

Android WebView是单进程架构，Compositor线程就是UI线程。因此要注意之前在Compositor线程上的逻辑现在是在UI线程上执行，要注意在UI线程上不应该存在一些耗时的操作比如图片解码等。

1.4 使用合适的功能线程

Chromium提供了一些功能线程供相关功能的代码执行，如文件操作必须在Chrome_FileThread线程上执行，数据库的操作必须在Chrome_DBThread线程上执行等等。

因此绝对不可以在UI线程上执行文件等操作，同时也不应该在Render进程（虽然是单进程模型，但是逻辑上的Render进程还存在）中执行。

1.5 不使用同步IPC消息

Chromium不允许UI线程向任何线程发送同步IPC消息，即使用了也会被Chromium在底层转换为异步。

原则上Chromium允许在其他线程上发送同步IPC，但出于避免线程卡死的考虑，要尽量避免使用。

1.6 避免类方法过多

某些类如AwContents方法太多太庞大，后续增加功能时注意尽量不要在其中添加，可以的话，尝试使用manager将代码分离出去。

1.7 代码设计风格

在代码设计上应该参考Chromium的设计风格，如命名，结构等，尽量保持风格一致。

1.8 设计合理的SDK接口

暴露给上层的接口要简洁合理，要考虑后续内核作为SDK（不仅仅是浏览器）输出，对方的使用是否便捷。

二.内核编码原则

1.编码规范

1. Native/Java代码以Blink/Chromium/Google/Android的编码规范为准。
2. 提交代码前必须使用自动化格式工具和编码检查工具，对要提交的代码进行格式化和检查通过后才能提交。
3. 代码检查时可能报告原生的代码有格式问题，我们不要修改。

最后，编码时一个简单的原则就是：

以周边的代码格式为准，如果不知道怎样的格式才正确，就在代码库里搜索一下就行了，所有的情况都有

2.新增Feature

2.1 宏规定

新内核对Vivo新增代码的格式做了新的规定，以方便代码阅读和后续升级。我们编码时除了遵守之前的编码规范外，还需要遵守以下规定:

1. Feature的代码必须添加到宏定义中，方便打开关闭与代码查找。宏定义的方式使用VIVO_FEATURE_XXX_YYY_ZZZ格式。
2. 修改原生代码时也需要将修改代码使用宏定义隔离，格式同上。
3. 新增的函数或者变量要放到Chromium的原生代码后面，Java的新增代码因为无法包装，所以应该使用注释与Chromium的代码分离。注释的格式使用// VIVO_FEATURE_XXX_YYY_ZZZ_BEGIN/END格式。
4. 对于不属于某个Feature的代码，如对原生代码的BugFix，则需要使用VIVO_BUGFIX(非移植自Chromium官方)/CHROMIUM_PATCH（移植自Chromium官方）格式的注释代码。
5. 新增不属于任何Feature的代码，也不是BugFix的代码，放置在VIVO_CHANGES宏定义中，这个宏是永远不会关闭的。

2.2 格式定义

新内核会在Native端和Java端使用Feature宏定义来分割vivo自己增加的feature代码，从而达到可以随时开关Feature的效果，方便调试和后续的升级工作。Native端和java端代码的宏定义应该遵守相应的编码规范。

Native端的宏定义形式应该为：

*VIVO**_FEATURE_XXX_YYY_ZZZ*

Java端的定义形式应该为：

vivoFeatureXxxYyyZzz

但是在Chromium中有一个例外，当使用BuildConfig时，则可以使用Native端宏的定义方式，即BuildConfig.VIVO_FEATURE_XXX_YYY_ZZZ的形式。

而我们的Java宏定义方式使用BuildConfig。要使用BuildConfig，需要：

import org.chromium.base.BuildConfig;

例如，图片解码内存优化的需求：

Native端宏定义为：

*VIVO**_FEATURE_IMAGE_DOWN_SAMPLE*

则native端宏定义包装feature代码的使用方法为：

*#if defined(**VIVO**_FEATURE_IMAGE_DOWN_SAMPLE)*

...

#endif

而Java端定义使用了BuildConfig，则也应该是：

*VIVO**_FEATURE_IMAGE_DOWN_SAMPLE*

则在Java端包装feature代码的使用方法为：

*if (**Build**Config.**VIVO**_FEATURE_IMAGE_DOWN_SAMPLE) {*

...

}

具体使用方式见:http://km.vivo.xyz/pages/viewpage.action?pageId=38228518

3.代码检查工具与格式化工具

3.1 代码检查工具code-check

code-check是基于cpplint开发了code-check脚本, 用于检查某个文件的格式是否符合谷歌的编码规范。

使用方法：

在src目录下，运行：

vivo/tools/code-check file_path_name

就可以对file_path_name指定的文件进行检查，工具会将检查出的问题输出到命令行。

我们可以根据提示，修改指定的问题。注意，如果代码本身的问题，则不需要我们修改。

3.2 代码检查工具code-format

code-format是基于clang-format开发的脚本，用于自动格式化本次提交所产生的diff文件。这个diff文件的生命周期为从本地修改代码开始到提交到远程仓库为止。

使用方法：

本地修改了代码后，在src目录下，运行：

vivo/tools/code-format

直接对修改的代码进行格式化，格式化不会影响到其他未修改的代码。

4.新增文件命名与路径

所有新增的文件都放到VIVO目录下，在VIVO目录下的层级要与Chromium中的层级一致。

命名：

举个例子，如标题栏控制类，设计时如果认为这个类的实现在AndroidWebView层的，那么命名上需要体现这个类的层级，因此合理的命名应该为AwTopControlsManager。

根据层级的不同，一般命名会加上对应的前缀：

BrowserSDK层：WebXXX， 如WebView， WebViewClient、WebVideoView。

Glue层：Glue层中的AOSP代码同样应该命名为WebXXX，如WebView、WebViewChromium。

AndroidWebView层：AwXXX，如AwContents、AwSettings。

Content层：ContentXXX，如ContentViewCore、ContentVideoView。

从AndroidWebView层开始有对应的Native代码，Native的命名与Java的命名方式相对应，这个参考Chromium就可以了。

回调类：Chromium中的回调在Java层多使用XXXClient，也有使用Listener的，根据实际情况具体分析命名方式，建议使用Client。

Native端：主要参照Native端的各个模块的命名规则，如Net、CC、Blink等，这里需要注意的一点是Browser端和Render端的概念，在命名上需要分清，Browser端命名为XXXHost、XXXHostImpl，

Render端命名为XXX、XXXImpl，大家注意一下。

文件路径：

Java文件

上面那个例子中的AwTopControlsManager， vivo目录中有一个Java文件夹，所有的Java文件都应该放在这里面。

这个新增文件对应的Chromium中AndroidWebView层上的路径为：

src/android_webview/java/src/org/chromium/android_webview

因此这个新增文件应该放置的路径是：

src/vivo/java/android_webview/src/com/vivo/android_webview/

Native文件

Native的路径也是参考Chromium的方式，直接在vivo目录下新建对应模块名字的文件夹，然后在对应的路径下添加文件。

这里举例为ImageDownSample.cpp。这个新增类对应的逻辑应该是Blink代码中的图片解码模块。图片解码模块在Chromium中的路径为：

src/third_party/Webkit/Source/platform/image-decoders/

因此这个新增文件应该放置的路径是：

src/vivo/third_party/WebKit/Source/platform/image-decoders

三.模块设计注意事项

1.多线程安全

多个线程操作同一个对象时，一定要注意线程安全问题。线程安全出现的根本原因：

1.存在两个或者两个以上的线程对象共享同一个资源；

2.操作共享资源代码有多个语句。

1.1 同步锁

加入了同步锁可以解决一部分问题，但是一些场景下安全隐患依然存在：

1. 如需要确认同步锁是否加在了需要被同步的代码块的位置上。
2. 如果同步代码的位置没有错误，这时就再看同步代码块上使用的锁对象是否是同一个。多个线程是否在共享同一把锁。很多时候发生问题，是因为加锁的对象不对，从而导致同步失效。

   原则：对象锁作用范围最小化，确保线程无死锁等

Good:

```
private final Object mProxyThreadLock = new Object();

private void stopProxyThread() {
    synchronized (mProxyThreadLock) {   // 同一把锁
        if (mProxyThread != null) {
            mProxyThread.stopServer();
            mProxyThread = null;
        }
    }
 }

private void startProxyService(String proxyIp) {
    ......
    synchronized (mProxyThreadLock) {   // 同一把锁
        mProxyThread = new ProxyThread(proxyIp, new ProxyToastCallback() {
            ......
            mProxyThread.setPort(mLocalPort);
            mProxyThread.startServer();
            .......
        }
    }
```

```
}
```

Bad:

```
// 示例1:
private void stopProxyThread() {
    WorkerThread.runOnWorkerThread(new Runnable() {
        @Override
        public void run() {
            synchronized(this) { // 锁对象不正确
                if (mProxyThread != null) {
                    mProxyThread.stopServer();
                    mProxyThread = null;
                }
            }
        }
    });
}

private synchronized void startProxyService(String proxyIp) {  // 锁对象不正确
    ......
     mProxyThread = new ProxyThread(proxyIp, new ProxyToastCallback() {
        ......
        mProxyThread.setPort(mLocalPort);
        mProxyThread.startServer();
        .......
    }
}
```

1.2 跨语法混用

对于可能出现多线程操作的场景要敏感，如果只是单纯的使用java或者C++进行编码还好，大家可能存在一定的警惕性。

但是如果涉及到java和C++混合使用的场合，这时候需要保持清醒，需要明确的分清楚哪些调用是必须UI线程使用，哪些操作是只能通过线程来异步处理。

如果是非UI线程的调用，且涉及到JAVA/C++数据传输的，请确认是否需要进行同步安全处理。

Good:

```
// 加上锁，保证同时只有1个线程能操作该对象
// 调用1:java中调用该接口处于非ui线程
void PersistentHostCache::Set(const Key& key,
                              const Entry& entry,
                              base::TimeTicks now,
                              base::TimeDelta ttl) {
  {
    base::AutoLock lock(access_lock_);   // 加上锁
    HostCache::Set(key, entry, now, ttl);
  }
```

```
  ......
}

// 调用2:c++中在网络线程中调用该接口
void HostResolverImpl::CacheResult(const Key& key,
                                   const HostCache::Entry& entry,
                                   base::TimeDelta ttl) {
  ......
  if (cache_.get())     // cache_.get()获的为HostCache指针对象
    cache_->Set(key, entry, base::TimeTicks::Now(), ttl);
}
```

Bad:

```
// 两个线程同时操作,产生多线程同步问题
// 调用1:java中调用该接口处于非ui线程
void PersistentHostCache::Set(const Key& key,
                              const Entry& entry,
                              base::TimeTicks now,
                              base::TimeDelta ttl) {
  HostCache::Set(key, entry, now, ttl);
  ......
}

// 调用2:c++中在网络线程中调用该接口
void HostResolverImpl::CacheResult(const Key& key,
                                   const HostCache::Entry& entry,
                                   base::TimeDelta ttl) {
  ......
  if (cache_.get())    // cache_.get()获的为HostCache指针对象
    cache_->Set(key, entry, base::TimeTicks::Now(), ttl);
}
```

1.3 多线程使用全局变量

多线程场景下使用全局变量时，需要先使用局部变量持有，避免使用时被全局变量引用被修改。

Good:

```
  mWaitTask = new TimerTask() {
  @Override
     public void run() {
         Message message = new Message();
         message.what = WAIT_TIME_OUT;
         Handler tmpHandler = mHandler; //使用局部变量持有全局对象,将数据保存下来
         if (tmpHandler != null) {      //对局部变量进行判空处理
              tmpHandler.sendMessage(message);
         }
     }
```

```
    };


     private void clear() {
     ......
         if (mWaitTask != null) { //先判断task不为空
              mWaitTask.cancel();
              mWaitTask = null;
          }

         if (mHandler != null) { //再对全局变量mHandler进行处理
              mHandler.removeCallbacksAndMessages(null);
              mHandler = null;
          }
     }
```

Bad:

```
    mWaitTask = new TimerTask() {
    @Override
       public void run() {
            Message message = new Message();
            message.what = WAIT_TIME_OUT;
            if (mHandler != null) {  //该线程对mHandler先判空了,但是clear这个线程可能接下来就将
mHandler置空了,从而导致异常
                 mHandler.sendMessage(message);
            }
       }
    };


     private void clear() {
     ......
         if (mHandler != null) { //对全局变量mHandler进行处理
              mHandler.removeCallbacksAndMessages(null);
              mHandler = null;
          }
         if (mWaitTask != null) {
              mWaitTask.cancel();
              mWaitTask = null;
          }
     }
```

总结：多线程场景下，尽量使用线程安全的对象进行数据操作，如vector,CopyOnWriteArrayList.

2.遍历安全

不要在循环里进行add/remove操作，remove元素请使用Iterator方式，如果并发操作，需要对Iterator对象加锁或者使用线程安全类型

Good:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String item = iterator.next();
    if (删除元素的条件) {
        iterator.remove();
    }
}
```

Bad:

```
// 反例1:
public synchronized void stop() {
    ......
    if(mRuleList != null && mRuleList.size() > 0) {
        List<Integer> indexList = new ArrayList<>();
        for (int i = 0; i < mRuleList.size(); i++) {
            AddressItem item = mRuleList.get(i);
            if (TextUtils.isEmpty(item.getCurrentIp())) {
                indexList.add(i);
            }
        }
        for (Integer index : indexList) {
            mRuleList.remove(index.intValue());
        }
    }
    ......
}

// 反例2
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
for (String item : list) {
    if ("1".equals(item)) {
        list.remove(item);
    }
}
```

2.数组越界

进行数组操作时，严格控制防止发生数组越界问题。

Good:

```
size_t pos = history_control_host_list_.find(host);

if (pos != std::string::npos && (
      pos == 0 || history_control_host_list_[pos-1] == '^')) {
   ......
}
```

Bad：

```
size_t pos = history_control_host_list_.find(host);

//[pos -1]很可能发生数组越界行为
if (pos != std::string::npos && (
      history_control_host_list_[pos-1] == '^')) {
   ......
}
```

3.内存泄漏

socket，file等句柄，使用完成后必须要执行关闭操作。

而文件句柄的关闭请在finally里进行操作，防止内存泄漏，主要可能的泄漏对象包括以下类型：java.io.FileInputStream java.lang.Process android.database.MatrixCursor java.io.FileOutputStream java.io.RandomAccessFile com.iqoo.secure.update.download.DownloadManager com.iqoo.secure.update.download.os.IndentingPrintWriter (java.io.PrintWriter) 具体参考规范定义： 维沃移动通信有限公司 Android-Java 编程规范 ---7.3.6 不在 finally 释放引起内存泄漏这一项。

Good:

```
int socket_fd = 0;

do {
  socket_fd = CreateSocket(); // create操作
  if (socket_fd < 0) {
    ......
    break;
  }

  close(socket_fd);  // close操作
  return 0;
} while (0);

close(socket_fd); // close操作
```

Bad:

```
do {
  CreateSocket(); // 有create,无close操作
  if (socket_fd < 0) {
    ......
    break;
  }
  return 0;
} while (0);
```

4.类型强转

进行element对象转换时，一定要经过类型检查，否则直接强制转换会导致异常

Good:

```
if (!AllowedToRequestFullscreen(document) && !isHTMLVideoElement(pending)) {
    ......
}
```

Bad：

```
if (!AllowedToRequestFullscreen(document) &&
!ToHTMLMediaElement(pending).IsHTMLVideoElement()) {
    // 直接将element强转成media element后，再进行video类型判断,这样导致了类型异常
    ......
}
```

5.兼容性处理

5.1 SDK兼容性设计

对外接口暴露及设计，需要考虑SDK兼容性。接入方不能绕过SDK层直接使用内核的类，这会导致接入方和内核的逻辑耦合。

类似在UI直接import内核文件，这类操作是严格禁止的。如：

Bad:

```
// UI包
package com.vivo.browser.ui.module.hstspreload;

// Wrong! 该文件为内核类，直接暴露给UI进行import了
import org.chromium.net.HSTSPreloadBridge;
import java.util.ArrayList;
```

5.2 可扩展设计

5.2.1 *大功能可服务器下发配置修改；*

- 涉及到大的功能点时，需要提供服务端配置下发功能，以便快速解决问题。
- 尽量复用已有的配置文件，禁止为每个功能都添加新的下发配置文件。
- 名单精准匹配能力。内核下发的各类名单功能，优先考虑是否具有具体url匹配的能力，而不是host模糊匹配。

Good:

```
public static void updateVideoBlackList(String key, String value) {
    if (WEBPAGE_VIDEO_STYLE_BLACK_LIST_KEY.equals(key)) {
        // 配置视频是否使用原生界面
        sWebpageVideoStyleBlackList = value.split("\\^");
    }
    ......
    else if (PLAYBACK_POSITION_CACHING_WHITE_LIST_KEY.equals(key)) {
        // 配置视频断点续播的开关
        sPlaybackPositionCachingWhiteList = value.split("\\^");
    }
}
```

5.2.2 数据库兼容

数据库升级，降级时，需要考虑数据兼容及稳定性。并按照实际情况，进行数据库增删改操作。

异常场景，以及版本升级等各种场景下必须要考虑出错或数据库兼容处理。

Good:

```
@Override
public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // 降级时，删除已有数据,防止发生崩溃。
    deleteAllTables(db);
    onCreate(db);
}
```

Good:

```
public class SharedPreferenceUtils {
    private static final String TAG = "SharedPreferenceUtils";

    // pref name
    public static final String KEY_DEFAULT_PREF = "";
    public static final String KEY_CORE_COMMON_PREF = "core_common_pref";
    public static final String KEY_PROXY_RULE_DEFAULT_PREF = "proxyrule_xml_default";
    //public static final String KEY_PROXY_RULE_PREF = "proxyrule_xml";
    // 新加的字段使用单独的版本标识
    public static final String KEY_PROXY_RULE_PREF = "proxyrule_xml_core_25";
```

5.3.PC和Mobile版兼容性

功能设计时，需要考虑到不同排版模式下，功能及页面展示是否正常。

如视频悬浮置顶功能，在分屏和PC版模式下不生效。

5.4 内外销兼容性

现内外销内核使用同一份代码，如涉及到内外销功能区别，如埋点，语言，media特性等请使用编译宏等方式来进行控制区分。

Good:

```
public static HashMap<String, String> appendUpgradeParams(HashMap<String, String>
params) {
    ......
    // 可用宏或者其他方式来进行控制内外销区别
    if (!BuildConfig.VIVO_FEATURE_BUILD_EX) {
        final String mieiCode = getImei(ContextUtils.getApplicationContext());
        final String modelNumber = getModelName();
        if (TextUtils.isEmpty(mieiCode) || INVALID_MIEICODE.equals(mieiCode)) {
            params.put(MIEI_CODE_TAG, "012345678987654");
        } else {
            params.put(MIEI_CODE_TAG, mieiCode);
        }
        params.put(MODEL_NUMBER_TAG, modelNumber);
    }
    ......
    return params;
}
```

注：多语言的适配提交，以后请全部使用工具进行提交

http://vivotranslation.vivo.xyz:8082/transtool/?spm=0.0.0.0.pzoFun

5.5 数据解析兼容性

- 数据解析时，需要考虑各种写法错误及笔误等情况，防止引起解析失败。

Bad:

```
std::vector<std::string> UserAgentManager::Split(std::string str,
                                                  std::string pattern) {
  std::vector<std::string> result;
  str += pattern;
  // 接下来对str进行解析处理
  // str执行了+pattern。但一旦下发资源时,最后一个不小心写了$等符号,这边解析结果可能出错。
  .......
```

- 数据解析匹配时，最好使用大小写不敏感的写法，防止因为html或者其他用户写法不标准，导致兼容性问题。

6.安全性设计

6.1 阻止webview通过file:schema方式访问本地敏感数据。

6.2 移动网络下严禁非用户预期的流量消耗。

如视频预加载，视频缓存等

6.3 敏感数据或网络接口请求参数加密

Good:

```
if (TextUtils.isEmpty(mUserID)) {
    mUserID = AESUtils.encrypt(SystemUtils.getImei(ctx)); //对IMEI信息进行加密处理
     ......
}
```

- 外销部分国家，部分数据如IMEI,国家码等直接不能上报或者传输，在适配外销过程中，需要时时警惕该部分数据。
- 本地日志输出，需要将IMEI，session等进行加密或者脱敏处理，保证不会泄漏用户隐私。

6.4 功能或函数默认接口返回值需要合理，可控。

如默认网络限制打开，防止后台或者意外跑流量。默认无法判断网络状态时，返回unknown或者无网络链接。

Good:

```
public boolean isNetRestricted() {
    return mNativeVivoContentVideoView != 0
             ? nativeIsNetworkRestricted(mNativeVivoContentVideoView)
            : true;    //意外或者无法判断的场景下,默认进行流量管控限制
}
```

6.5 优先使用时间戳进行时间计算

使用定时器，时间差值等逻辑计算时，尽量使用timetick进行计算，而不是system时间点，因为system时间点存在被恶意更改的可能性，会导致逻辑异常，存在漏洞。

Good:

```
 base::TimeTicks now_time = base::TimeTicks::Now();
  if (connection_type != NetworkChangeNotifier::CONNECTION_NONE &&
      !last_connection_change_time_.is_null()) {
    base::TimeDelta time_delta = now_time - last_connection_change_time_;
    ......
  }
```

Bad：

```
  base::Time now_time = base::Time::NowFromSystemTime(); // 用户或者脚本修改了系统时间就会导致异常
  if (connection_type != NetworkChangeNotifier::CONNECTION_NONE &&
      !last_connection_change_time_.is_null()) {
    base::TimeDelta time_delta = now_time - last_connection_change_time_;
    ......
  }
```

6.6 WebView绕过证书校验漏洞

WebView组件加载SSL书认证发生错误时，会调用WebViewClient类的onReceivedSslError方法 .

当证书出现问题的时候，有 2 种常见场景。

- handler.cancel() -- 系统默认不加载该网页

```
    public class WebViewClient {

        public void onReceivedSslError(WebView view, SslErrorHandler handler,
                SslError error) {
            handler.cancel();
        }
        ...

    }
```

- handler.proceed()

  如果该方法实现调用了handler.proceed()来忽略该证书错误，则会受到中间人攻击的威胁，可能导致隐私泄露。

可行的解决方案：

把服务器的证书放置在 assert 文件夹，当出现 ssl error 的时候进行读取，然后与服务器校验，校验通过了就加载该网页。校验不通过，不打开网页，进行安全提醒。

7.稳定性设计

7.1 基于稳定性和SDK兼容性考虑，内核新加功能默认开关必须关闭。

7.2 做为内核提供给上层的接口，需要考虑返回数据合理性及稳定性。

如：返回值会参数是否会出现异常崩溃，ANR等问题。

当返回值或参数长度过大时，很可能引起ANR问题。故看图模式等功能返回时，应过滤base64编码的数据，不做base64编码的图片的相关处理。

7.3 JNI调用保护

Java调用C++方法前(JNI)，需要添加destory判断，防止产生webview被销毁后的调用异常。

if (isDestroyedOrNoOperation(WARN)) return;

```
public void acquireImageData(String url) {
    if (isDestroyedOrNoOperation(WARN)) return; // 防止java和c++webview状态不统一,导致异常
    if (BuildConfig.VIVO_FEATURE_PICTURE_MODE)
nativeAcquireImageData(mNativeAwContents, url);
}
```

7.4 上层适配能力支持

内核开发新功能时，需要具备提前识别各种风险的能力。

包括不仅限于：上层适配风险，用户体验问题，接口使用风险，广告收入问题（埋点如何设计，埋点如何用，广告收益最大化时内核需提供的支撑能力等）

7.5 字符窜匹配

String匹配，查找等处理，尽量使用常量来进行equal等对比操作，因为传入的string参数为空,会导致崩溃问题。

Bad:

```
public void setCurrentProxyTypeString(String proxyTypeString) {
    LogUtils.i(TAG, "setCurrentProxyTypeString " + proxyTypeString);
    mCurrentProxyTypeString = proxyTypeString;

    if (proxyTypeString.equalsIgnoreCase(PROXY_MAA) ||
proxyTypeString.equalsIgnoreCase(PROXY_MIXED)) {   // proxyTypeString为空,即会出现空指针异
常
    ......
    }
}
```

7.6 新功能实现时，不能为了实现的简单或者调用方式的简单，破坏原有的代码架构及设计。需要通盘考虑下功能的设计是否合理，正确。

涉及到数据更新等操作，最好使用注册监听或者主动通知的方式，不能为了调用简单，直接跨线程使用，应避免多线程操作引起的数据混乱或者数组越界，对象为空，数据销毁等异常现象

8.共享内存的安全使用

对于超大尺寸图片,如果未注意其解码后数据量大小，会导致共享内存释放不及时,最终耗光共享内存，引发浏览器崩溃。理论上，一般共享内存中的单张图片消耗不要超过10M。如果需要计算解码后图片占用内存可参考如下两个公式。

不同的场景下，请注意合理的估算方式。如解码前，使用公式一估算即可，如已解码，请使用公式二进行估算。

公式一：

解码后数据量 = 图片宽 * 图片高 * 4

公式二：

解码后数据量 = 图片.size()

关于图片解码相关知识，请参考：

http://192.168.2.22/repositories/DocsAndTools/Docs/SW-DOC/互联网软件开发部/互联网应用组/浏览器/开发文档/渲染文档/STD&SPD*软件方案设计*浏览器图片解码内存优化v1.0.docx

Good:

```
static size_t kMaxPictureModeSize = 1024 * 1024 * 10; //定义最大的阈值

bool RenderViewImpl::CopyImageDataToSharedMem(
    const blink::WebData& image_data,
    base::SharedMemoryHandle* shared_mem_handle) {
  uint32_t buf_size = image_data.size();
  if (buf_size == 0 || buf_size > kMaxPictureModeSize) //对解码后的图片进行size判断
    return false;

  std::unique_ptr<base::SharedMemory> shared_buf(
      content::RenderThread::Get()->HostAllocateSharedMemoryBuffer(buf_size));
  if (!shared_buf)
    return false;

  if (!shared_buf->Map(buf_size))
    return false;
```

9.Handler的安全使用

Handler对象与其调用者在同一线程中，如果在Handler中设置了延时操作，则调用线程也会堵塞。每个Handler对象都会绑定一个Looper对象，每个Looper对象对应一个消息队列（MessageQueue）。如果在创建Handler时不指定与其绑定的Looper对象，系统默认会将当前线程的Looper绑定到该Handler上。

在主线程中，可以直接使用new Handler()创建Handler对象，其将自动与主线程的Looper对象绑定；

在非主线程中直接这样创建Handler则会报错(Can't create handler inside thread that has not called Looper.prepare()

)，因为Android系统默认情况下非主线程中没有开启Looper，而Handler对象必须绑定Looper对象。

Good:

```
protected ServerData(String dataName) {
    mDataName = dataName;
    mRawJSON = "";

    mRetryTimes = 0;
    mHandler = new Handler(Looper.getMainLooper()); //非主线程中创建handler时，注意绑定合适的
looper对象
    ......
}
```

9.数据解析

请严格按照编码规范进行代码开发，不要随性或者个人偏好来处理。有公共工具类的，请使用工具类处理，如JSON的解析请全部使用JsonParserUtils来处理。JsonParserUtils已经对各种空指针，内容为空等异常做过兼容处理。自行解析的时候，容易忽略空指针，数据返回为空等异常场景。

10.公共代码逻辑请抽象

公共代码逻辑请抽象成基类，严禁非必要的代码冗余。如果同样的代码逻辑存在多处，很容易造成修改的遗留，带来隐形的bug隐患。比如：视频模块的全屏，小屏，v粉卡等各种模式，需要抽取公共基类进行处理。

11.日志输出

所有上线 log 请使用 VIVOLog或者 LOG(INFO) 这种类型。内核已重新定义上述两种log，将默认给其 tag 加上 git commit id, 方便定位线问题的代码与符号。

不允许直接使用 __android_log_XXX 或 log.X 这种裸 api。

四.SDK设计注意事项

1.SDK层中仅提供转接逻辑，不能出现具体实现。具体的实现放在内核中。

2.实现接口时要考虑线程要求，是否要求必须在UI线程上使用。

SDK接口中目前不提供线程检查方法，但内核中提供CheckThread检查，同时支持转抛到UI线程的方法。

3.内核中各种Context的使用方法

3.1 ApplicationContext

ApplicationContext可以理解为浏览器应用的Context，生命周期和浏览器相同，也可以从这个Context中获取资源和系统应用，但如果Activity中的Context有自己特别的处理的话，会与这个Context有些差别。

ActivityThread.currentApplication().getApplicationContext() 这个方法获取的就是Application的Context。

ContextUtils.getApplicationContext()方法获取到的是chromium封装的Context，与Application的Context稍有区别，对于我们一般使用来说，可以认为是Application的Context。

3.2 Activity Context

Activity的Context，也就是WebViewAdapter中getContext_outer中获取到的Context，是浏览器MainActivity的Context，一般情况下，与Application的Context差别不大，但生命周期有所不同，比如使用Service的情况，Application的Context与这个Context会有所区别。

resourcesContextWrapper方法获取到的Context是chromium和SDK共同封装的Context，与MainAcitivity的Context的区别在于，资源的定位路径不同。如果在内核中构建一些View，如弹窗等，一般使用这个Context，如果这些View用到了系统资源，不能使用这个。

总结一下：

1)ActivityThread.currentApplication().getApplicationContext() 这个方法获取的就是Application的Context。

2)ContextUtils.getApplicationContext()方法获取到的是chromium封装的Context，与Application的Context稍有区别，对于我们一般使用来说，可以认为是Application的Context。

3)为了避免风险，在不需要使用内核资源的情况下，建议大家使用ActivityThread.currentApplication().getApplicationContext()

4)若设计到内核资源，则使用上层 Context 访问,这时候需要对Context进行内核ResourceWrapper注入操作。

4.SDK扩增接口使用

1.内核版本<2.8时，所有新加的方法必须添加进BuildInfo.java.否则SDK无法判断是否能使用。这个需要所有内核同学进行规范化操作处理。

2.内核版本>=2.8时，需使用注解的方式，具体使用方式如下：

Requires：

对方法的注解，控制方法的限制使用内核版本，或者方法的使用宿主；

SDK会对每一个代理的方法进行检测，当配置的接口方法没有配置此注解，并且对上层暴露，上层使用时会引起Runtime异常

MirrorSource：

对类的注解，表明一个接口类只是另一个类的镜像；当SDK对内核的接口方法参数与SDK对上层的接口方法参数发生不一致时，SDK需要重新生成一份对上层的接口以隐藏内核接口的具体方法类型；

新生成的接口类需要加上此注解，以忽略SDK对Requires注解的检查；

CantNull:

对参数的注解，当对参数加入此注解时，上层传入控制时，SDK会对本次方法的调用进行忽略；

Translate：

对 参数的注解，表明当前参数与内核的参数类型不一致，SDK会在调用到内核时根据指定的类型进行转换；

ParamCheck：

对方法的注解，当对参数加入CantNull，Translate注解时，必须对方法加入本注解，否则会失效；

具体代码注解见：http://km.vivo.xyz/pages/viewpage.action?pageId=26883682

5.Android 6.0以下机型不能接入我们webview SDK，请提前跟接入方，明确该风险。

6.内核接口支持的功能及支持的范围必须明确标识，如看图模式AcquireImageDataF接口支持哪些格式进入看图模式，需要在接口中明确说明。如不支持base64和svg等图片格式。

7.部分控件访问android原生资源会发生失败的情况，如发生该问题，则不从xml加载布局，需要动态创建布局控件。

五.C++/JAVA编码注意事项

C++注意事项

1.头文件

1.使用#define防止多重包含

所有头文件都应该使用#define 防止头文件被多重包含，命名格式应当是：*H* 为保证唯一性，头文件的命名应基于其所在项目源代码树的全路径。例如，项目 foo 中的头文foo/src/bar/baz.h 按如下方式保护：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif
```

2.减少头文件依赖

使用前置声明尽量减少.h 文件中#include 的数量。

当一个头文件被包含的同时也引入了一项新的依赖，只要该头文件被修改，代码就要重新编译。如果你的头文件包含了其他头文件，这些头文件的任何改变也将导致那些包含了你的头文件的代码重新编译，因此，我们宁可尽量少包含头文件。

举例：头文件用到类File,但是不需要访问File的声明，则在头文件中前置声明class File即可。

在头文件如何做到使用类 File而无需访问类的定义？

1) 将数据成员类型声明为 File*或 File&;

2) 参数、返回值类型为 File的函数只是声明（但不定义实现）；

3) 静态数据成员的类型可以被声明为 File，因为静态数据成员的定义在类定义之外。

另一方面，如果你的类是 File的子类，或者含有类型为 File的非静态数据成员，则必须为之包含头文件。

2.类

1.为多态基类声明virtual析构函数

virtual函数的目的是允许继承类的实现能定制化。只有当class内至少有一个virtual函数时，将它声明为virtual析构函数

示例：

Good:

```
class Timekeeper {
    public:
        TimeKeeper();
        virtual ~Timekeeper();
};
```

Bad:

```
class Timekeeper {
    public:
        TimeKeeper();
        ~Timekeeper();
};

class AtomicClock: public TimeKeeper {...}
class WaterClock: public Timekeeper {...}

TimeKeeper* getTimeKeepr();
TimeKeeper* ptk = getTimeKeeper();
delete ptk; // 非虚析构函数,delete会造成局部销毁，从而导致内存泄漏
```

注1：纯虚函数，继承者必须实现该函数功能，但是纯虚函数基类可以有默认实现。

注2：虚函数，继承者可以实现该函数功能，也可以不实现。不实现默认采用基类实现逻辑。

2.绝不重新定义继承而来的缺省参数值

virtual函数是动态绑定的，而缺省参数值却是静态绑定。

Good：

```
class Shape {
    public:
        enum ShapeColor { Red, Green, Bule };
        void draw(ShapeColor color = Red) const {
            doDraw(color);
        }
    private:
        virtual void doDraw(ShapeColor color) const = 0;
};
class Rectangle: public Shape {
    public:
        ......
    private:
        virtual void doDraw(ShapeColor color) const;
};
```

Bad：

```
class Shape {
    public:
        enum ShapeColor { Red, Green, Bule };
        virtual void draw(ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape {
    public:
        virtual void draw(ShapeColor color = Green) const;
};

// 示例
Shape* ps;
Shape* pr = new Rectangle;
pr->draw();  // 调用的是Rectangle::draw(Shape::Red); 而不是Gren
```

3.为避免隐式转换，需将单参数构造函数声明为explicit

```
class B {
    public:
        explicti B(int x = 0, bool b = true);
};

void doSomething(B object); // 声明1个doSomething的函数

B bObj1;
doSomething(bObj1); // OK!

B bObj2(28); // OK!
doSomething(28);  // Wrong! 需要传入的是B对象而非int
```

3.其他C++特性

1.尽量使用const，enum，inline而非#define

Good:

```
Template<typename T>
    inline void findMaxValue(const T&a, const T&b) {
    f(a > b ? a : b);
}
```

Bad:

```
#define FIND_MAX_VALUE(a,b)  f((a) > (b)  ?  (a) : (b))

int a = 8;
int b = 3;
FIND_MAX_VALUE(++a, b);  // a被累加2次
FIND_MAX_VALUE(++a, b+8);// a被累加1次
```

2.使用const修饰，防止意外或无意义的赋值

示例:

```
const T operator*(const T& leftValue, const T& rightValue);
T a,b,c;

if (a * b == c) // ok!
if (a * b = c)  // 编译报错，加上const会防止意外赋值改变const对象
```

3.以对象管理资源，尽量使用智能指针

3.1 auto_ptr

Good:

```
void f() {
    std::auto_ptr<A> pa(new A(123));
    ...
}
```

Bad:

```
void f() {
    A* pa = new A(123);
    // return 或异常会导致delete未执行,造成内存泄漏
    ...
    delete pa;
}
```

auto_ptr存在两个缺点

1.缺乏拷贝语义

c++中接触到的拷贝一般有三种，深拷贝，浅拷贝，转移拷贝。

a、浅拷贝：

就是逐字节拷贝，默认拷贝构造函数就是浅拷贝。直接将一个指针赋值给另一个指针，导致两个指针指向同一块内存。

b、深拷贝

一个指针赋值给另一个指针的同时，进行了内存的处理，两个指针有自己独立的内存空间。

c、转移拷贝

一个指针赋值给另一个指针之后，前一个指针（源指针被赋值为空）。

auto_ptr使用的是转移语义的拷贝。

```
void test(){
    auto_ptr<A> pa(new A(123));
    pa->print();

    auto_ptr<A> pb = pa; // 拷贝构造
    pb->print();
    pa->print();   // 段错误 pa为空指针
}
```

2.不能管理对象数组

```
void test(){
    // Wrong! auto_ptr析构函数中只有delete，无delete[]操作。
    auto_ptr<A> pa(new A[3]);
}
```

3.2 shared_ptr

shared_ptr可以解决转移拷贝的问题，但依然无法管理对象数组

```
void test(){
    shared_ptr<A> pa(new A(123));
    pa->print();
    shared_ptr<A> pb = pa; // 拷贝构造
    pb->print();
    pa->print();   // pa能正常输出，并指向同一个对象
}
```

3.3 scoped_ptr

scoped_ptr是一个类似于auto_ptr的智能指针，它包装了new操作符在堆上分配的动态对象，能够保证动态创建的对象在任何时候都可以被正确的删除。但是scoped_ptr的所有权更加严格，不能转让，一旦scoped_pstr获取了对象的管理权，你就无法再从它那里取回来。

正如scoped_ptr(局部指针)名字的含义：这个智能指针只能在作用域里使用，不希望被转让。

scoped_ptr和auto_ptr的根本区别在于所有权。auto_ptr特意被设计为指针的所有权是可以被转移的，可以在函数之间传递，同一时刻只能有一个auto_ptr管理指针。而scoped_ptr把拷贝构造函数和赋值函数都声明为私有的，拒绝了指针所有权的转让，只有scoped_ptr自己能够管理指针，其他任何人都无权访问被管理的指针，从而保证了指针的绝对安全。

```
auto_ptr<int> ap(new int(10)); // 一个int自动指针
scoped_ptr<int>sp (ap);    // 从auto_ptr获得原始指针
assert(ap.get() == 0);     // 原auto_ptr不再拥有指针

ap.rest(new int(20));      // auto_ptr拥有新的指针

auto_ptr<int>ap2;
ap2 = ap;                 // ap2从ap获得原始指针，所有权转移
assert(ap.get() == 0);    // ap不再拥有原始指针
scoped_ptr<int>sp2;       // 另一个scoped_ptr
sp2 = sp;                 // 赋值操作,无法同过编译!
```

Google建议：如果确实需要使用智能指针的话，scoped_ptr 完全可以胜任。在非常特殊的情况下，例如对 STL 容器中对象，你应该只使用 std::shared_ptr，任何情况下都不要使用auto_ptr。

3.4 unique_ptr

在Chromium里最常用的两种智能指针类型是`std::unique_ptr<>`和`scoped_refptr<>`。前者适用于单一所有权的对象，后者适用于引用计数的对象（然而，通常应该避免使用引用计数的对象）

`std::unique_ptr<>`持有对对象的独有权——两个unique_ptr不能指向一个对象，不能进行复制操作只能进行移动操作。

```
//示例1
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1;  // 编译会出错
```

```
std::unique_ptr<int> p3 = std::move(p1); // 转移所有权, 现在那块内存归p3所有, p1成为无效的针.
std::unique_ptr<int> p4(p1);  // 编译会出错
p3.reset();  //释放内存.
p1.reset();  //无效

//示例2
void Foo(std::unique_ptr<Bar> bar);
...
std::unique_ptr<Bar> bar_ptr(new Bar());
Foo(std::move(bar_ptr));          // 语句执行完后, |bar_ptr| 为 null.
Foo(std::unique_ptr<Bar>(new Bar()));
```

3.5 WeakPtr

`WeakPtr<>` 实际上不是智能指针。它的表现像指针类型，但是并不能用来自动释放对象，通常用作追踪其它地方拥有的对象是否依然存活，当追踪对象释放时，`WeakPtr<>` 会自动的置为 `null`。（但是依然需要在解引用前判断是否为 `null`，因为解一个 `null WeakPtr<>` 引用等于于解引用 `null`，而不是no-op。） `WeakPtr<>` 与C++11的 `std::weak_ptr<>` 作用比较相似，但是使用了不同的API并且少了许多使用限制。

3.6 RefPtr

RefPtr的思路很简单，自动地在各项操作中加上deref和ref,看看它最关键几个方法的源码：

```
template<typename T> inline RefPtr<T>& RefPtr<T>::operator=(T* optr)
    {
        refIfNotNull(optr);
        T* ptr = m_ptr;
        m_ptr = optr;
        derefIfNotNull(ptr);
        return *this;
    }

ALWAYS_INLINE RefPtr(T* ptr) : m_ptr(ptr) { refIfNotNull(ptr); }
```

示例：

```
class Document {
    ...
    RefPtr<Title> m_title;
}

void Document::setTitle(Title* title)
{
    m_title = title;
}
```

3.7 PassRefPtr

PassRefPtr主要用于参数传递中，当传递完成后，被PassRefPtr包装的对象就会被销毁，并且整个过程中不改变对象引用 。最终效果就是Node的引用为1，并且中间没有引用的变化。但是PassRefPtr是不能替代RefPtr的，因为被赋值后，它就是NULL了，再调用就会有空指针的错误。

3.8 内核开发中如何选择使用哪种智能指针?

- **单一所有权的对象**。使用`std::unique_ptr`。需要注意的是，`std::unique_ptr`持有的需要必须是非引用计数的，并且分配在堆上的对象。
- **无所有权的对象**。使用裸指针或者`WeakPtr<>`。注意`WeakPtr<>`只能在创建它的线程解引用（通常使用`WeakPtrFactory<>`）。如果你需要在对象释放前后立刻执行某些操作，那么可能使用callback或notification更适合，而不是`WeakPtr<>`。
- **引用计数的对象**。使用`scoped_refptr<>`，但是最好是重新考虑使用引用计数对象是否合理。引用计数对象很难明确拥有权和析构顺序，特别是在多线程环境中。总是有方法来重新设计引对象层级来避免引用计数的。限制每个类都只能在单个线程工作，并且使用`PostTask()`确保调用在正确的线程，这样有助于在多线程中避免引用计数。`base::Bind()`，`WeakPtr<>`等工具具备在对象释放时自动取消方法调用的能力。Chromium中依然有许多代码在使用引用计数对象，如果你看见Chromium中有代码这样做但并不代表这是合理的解决方案。
- **平台特定类型**。使用平台特定的对象，譬如`base::win::ScopedHandle`，`base::win::ScopedComPtr`，或者`base::mac::ScopedCFTypeRef`。需要注意的是这些类型使用方式可能和`std::unique_ptr<>`不同。

4.尽量使引用传递，而非值传递

除了内置类型和STL的迭代器，函数外（如int），一般而言引用传递效率更高，避免非必要的对象创建消耗。

按引用传递的参数必须加上 const。

```
void Foo(const string &in, string *out);
```

输入参数为值或常数引用，输出参数为指针；输入参数可以是常数指针，但不能使用非常数引用形参。

5.尽量少做转型操作

如果必须要进行转型，请使用C++新式转型。

1.普通转型

```
(T) experssion
```

2.新式转型

```
const_cast<T> (experssion)       // 移除常量属性

dynamic_cast<T> (experssion)     // 安全向下转型（Google建议除测试外不要使用）

reinterpret_cast<T> (experssion) // 指针类型和整型或其他指针间不安全的相互转换

static_cast<T> (experssion)      // 强迫隐式转换，如非const转const，父类向子类转换
```

Good:

```
class Window {
  public:
    virtual void onResize() {...}
    ...
};

class SpecialWindow: public Window {
  public:
    virtual void onReize() {
        Window::onResize();  // 调用Window::onReize作用于*this
        ...
    }
};
```

Bad：

```
class Window {
  public:
    virtual void onResize() {...}
    ...
};

class SpecialWindow: public Window {
  public:
    virtual void onReize() {
        static_cast<Window>(*this).onResize();
        // *this对象的基类部分的副本上调用了Window::onReize
        // 很可能造成基类部分数据并未更改，而继承部分数据被更改了,使得对象被局部变更
        ...
    }
};
```

6.变长数组和 alloca

禁止使用变长数组和 alloca()。

优点：变长数组具有浑然天成的语法，变长数组和 alloca()也都很高效。

缺点：变长数组和 alloca()不是标准 C++的组成部分，更重要的是，它们在堆栈（stack）上根据数据分配大小可能导致难以发现的内存泄漏："在我的机器上运行的好好的，到了产品中却莫名其妙的挂掉了"。

结论： 使用安全的分配器（allocator），如 scoped_ptr/scoped_array。

7.局部变量/成员变量安全使用

1）慎用同名的局部变量! 很容易产生局部覆盖全局变量等意外情况。

2）使用前，请明确变量声明及定义是否真实有效。防止无效使用：

Bad：

```
  int result = 0;
  int socket_fd = 0;

  do {
    socket_fd = CreateSocket();  // 1.给赋值socket_fd

    // fail will return -1
    if (socket_fd < 0) {         // 2.根据socket_fd值进行逻辑处理
      *os_error = VIVO_ERR_DNS_CREATE_SOCKET;
      break;
    }

    vivo_resolve_socket_fd_ = result; //3.本意是将socket_fd赋值给vivo_resolve_socket_fd_。但是使用了一个默认的result值，导致后续逻辑完全错误

    result = PrepareSocketBeforeRequest(vivo_resolve_socket_fd_);
    if (result < 0) {
      *os_error = VIVO_ERR_DNS_SETOPT_SOCKET;
      break;
    }
```

JAVA注意事项

1.java基础

1.迭代器

基于for方式的遍历，在遍历过程中**不允许对list进行修改**，否则会抛ConcurrentModificationException.正确的方式应该采用迭代器进行remove操作。

Bad:

```
for(String s:list){

if(s.equals("b")){

list.remove(s);    //报错

list.add(s);       //报错

}
```

迭代器部分使用注意事项如下：

1.1 如果迭代器的指针已经指向了集合的末尾，那么如果再调用next()会返回NoSuchElementException异常

Bad:

```
while (it.hasNext()) {

String next = (String) it.next();

If ("1".equals(next) {

it.remove();

}

}

// 迭代器的指针已经指向了集合的末尾

String next = (String) it.next();      //报错

// java.util.NoSuchElementException
```

1.2如果调用remove之前没有调用next是不合法的，会抛出IllegalStateException

Bad:

```
while (it.hasNext()) {

// 调用remove之前没有调用next是不合法的

 it.remove();        //报错

// java.lang.IllegalStateException

}
```

1.3 Iterator在迭代时，只能对元素进行获取(next())和删除(remove())的操作