IOS VS Android

本次分享无任何恶意，请android同学....

问题

- 为什么IOS比android使用更少的内存？
- 为什么IOS比android更流畅？
- 为什么IOS比android更安全？

目录

- IOS的智能后台模式
- IOS的内存管理模式
- IOS APNs
- IOS的沙盒原理
- IOS的签名原理

IOS的后台模式

- Android真后台，允许app留存在内存，依靠service服务组件继续运行
- IOS伪后台（智能后台）

IOS后台趋势历程

普通功能

IOS 3.x以下

无后台仅推送

IOS4 后

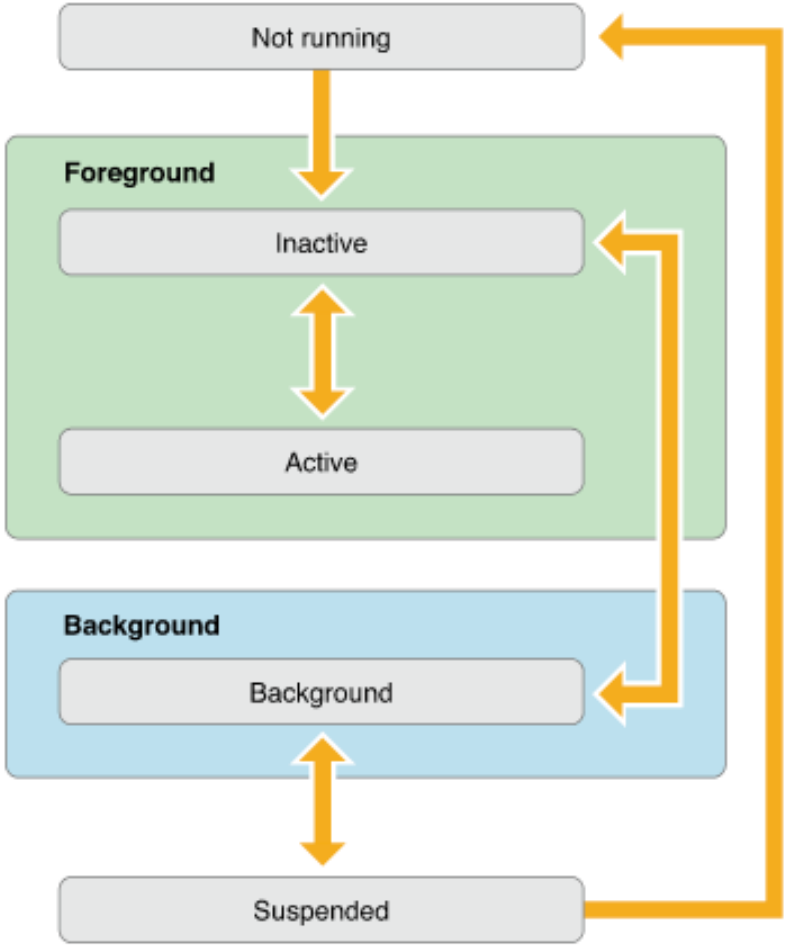
墓碑式
（进入后台被冻结）

IOS7 后

智能调度后台

部分功能真后台

1.后台音频
2.后台定位
3.后台语音服务
4.后台上传下载

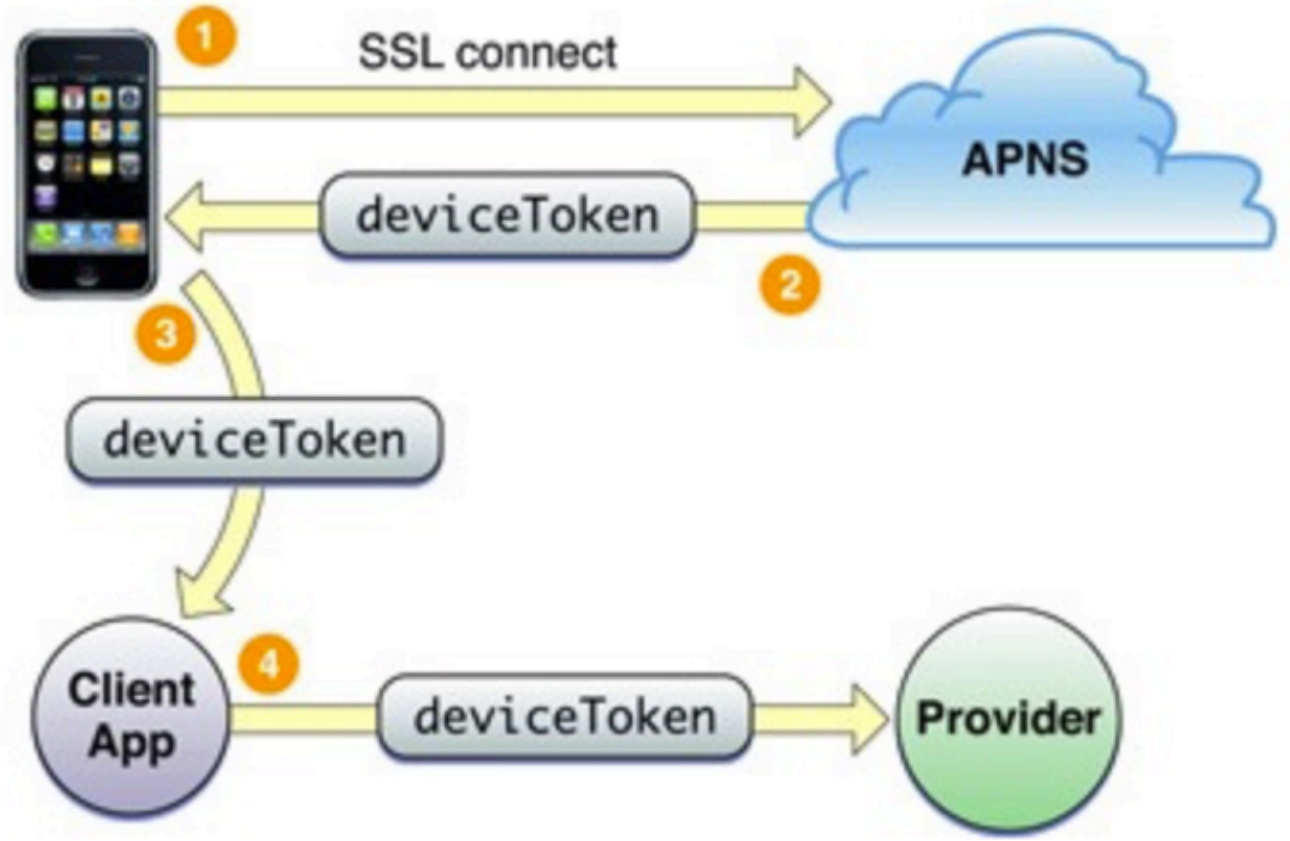


思考

- Ios为什么把后台模式搞得如此麻烦？

伪后台如何获取实时消息

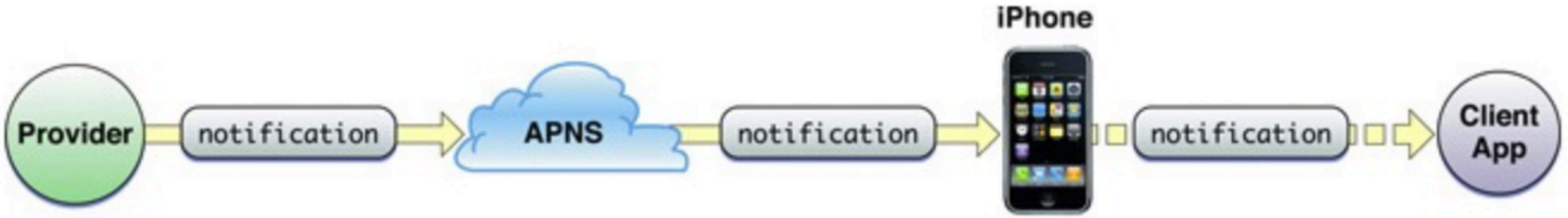
- APNs(App Push Notification Service)

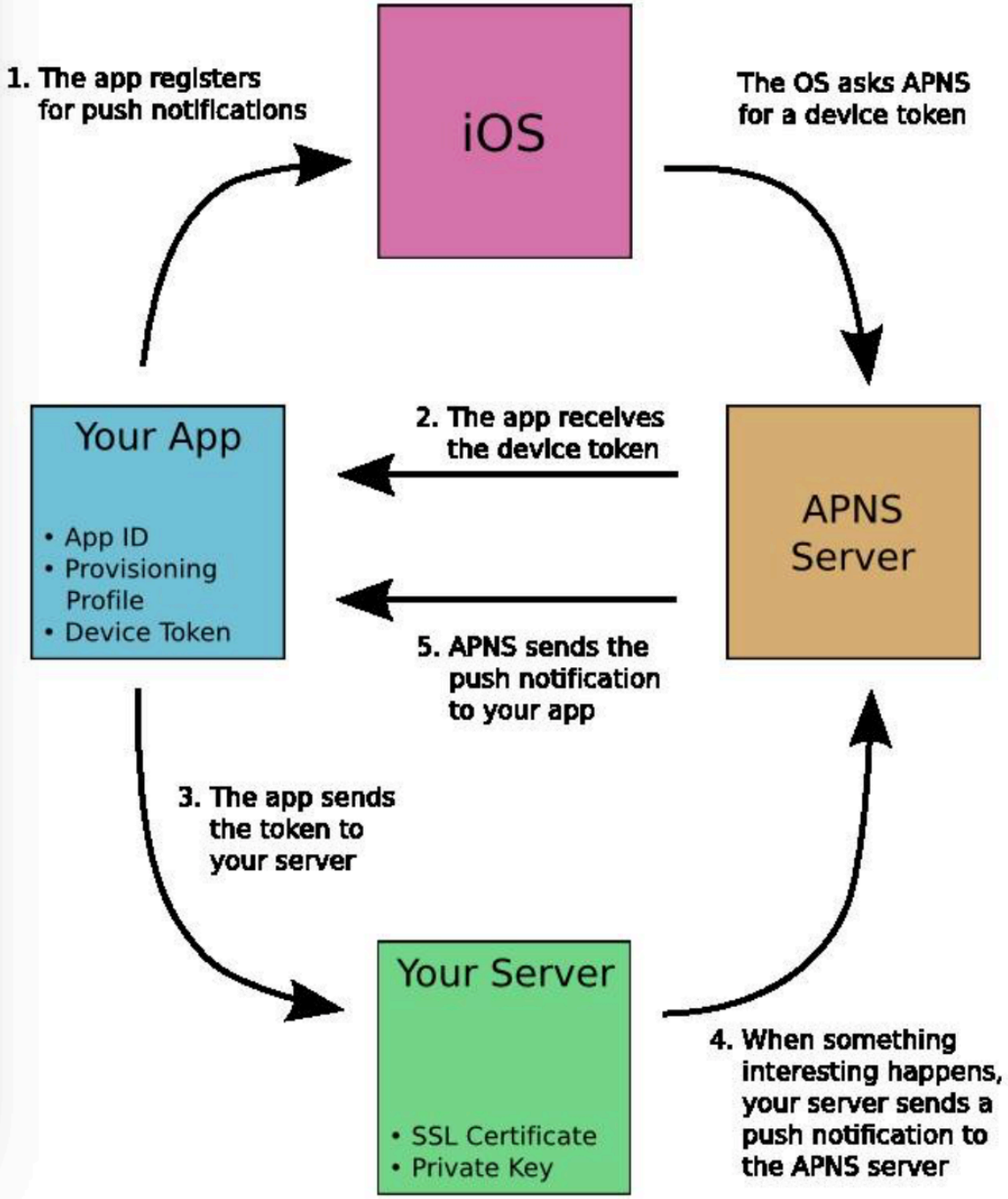


Token获取

推送流程

```
Bag Attributes
    friendlyName: Apple Development IOS Push Services: com.wuba.banghuangye
    localKeyID: 72 B7 5B E9 81 EE 4C 48 DB 91 F2 EE 47 5A 75 EB E6 30 43 29
subject=/UID=com.wuba.banghuangye/CN=Apple Development IOS Push Services:
com.wuba.banghuangye/OU=EFH63YDHFL/C=US
issuer=/C=US/O=Apple Inc./OU=Apple Worldwide Developer Relations/CN=Apple
Worldwide Developer Relations Certification Authority
-----BEGIN CERTIFICATE-----
```

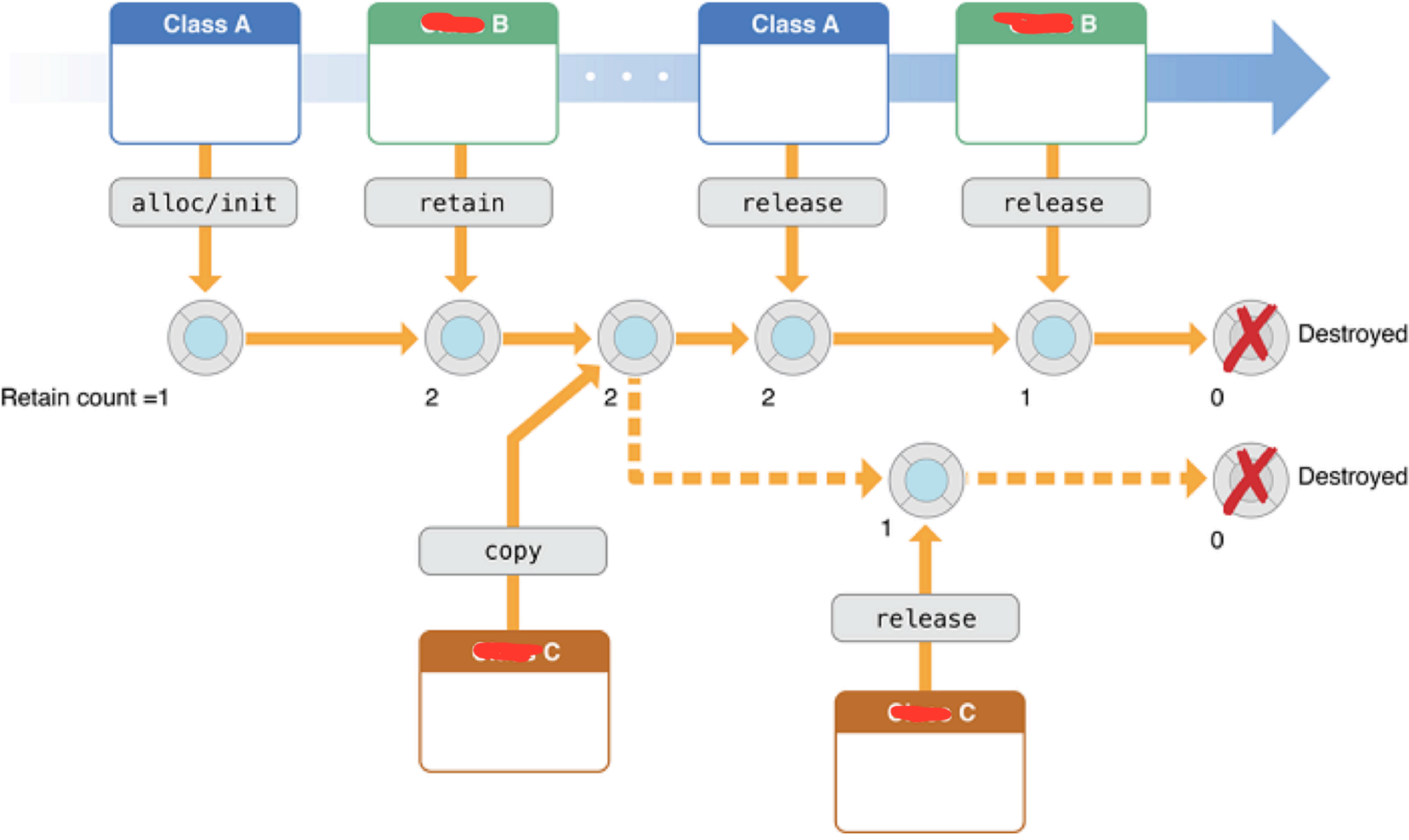



1. The app registers for push notifications
iOS
The OS asks APNS for a device token
Your App
• App ID
• Provisioning Profile
• Device Token
2. The app receives the device token
APNS Server
5. APNS sends the push notification to your app
3. The app sends the token to your server
Your Server
• SSL Certificate
• Private Key
4. When something interesting happens, your server sends a push notification to the APNS server

IOS内存管理模式

- MRC
- ARC及实现原理

IOS内存管理

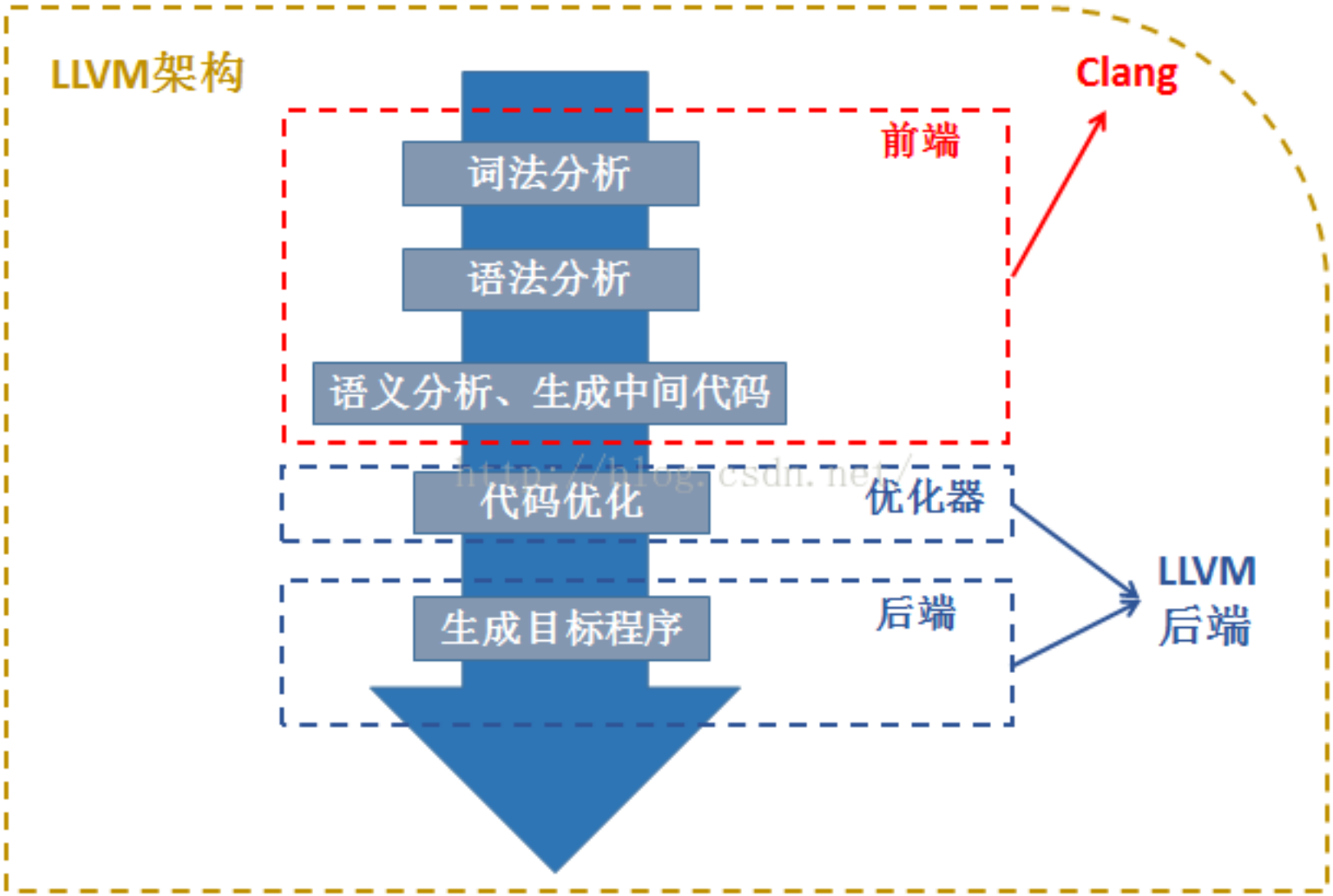

Class A
B
Class A
B
alloc/init
retain
release
release
Retain count =1
2
2
2
1
0
Destroyed
copy
C
1
0
Destroyed
release
C

MRC(**Manual Reference Counting**)

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // 只要创建一个对象默认引用计数器的值就是1
        Person *p = [[Person alloc] init];
        NSLog(@"retainCount = %lu", [p retainCount]); // 1

        // 只要给对象发送一个retain消息, 对象的引用计数器就会+1
        [p retain];

        NSLog(@"retainCount = %lu", [p retainCount]); // 2
        // 通过指针变量p,给p指向的对象发送一条release消息
        // 只要对象接收到release消息, 引用计数器就会-1
        // 只要一个对象的引用计数器为0, 系统就会释放对象

        [p release];
        // 需要注意的是: release并不代表销毁\回收对象, 仅仅是计数器-1
        NSLog(@"retainCount = %lu", [p retainCount]); // 1

        [p release]; // 0
        NSLog(@"--------");
    }
//    [p setAge:20];    // 此时对象已经被释放
    return 0;
}
```

ARC

```
int main(int argc, const char * argv[]) {
    // 不用写release, main函数执行完毕后p会被自动释放
    Person *p = [[Person alloc] init];

    return 0;
}
```


LLVM架构
Clang
前端
词法分析
语法分析
语义分析、生成中间代码
代码优化
优化器
生成目标程序
后端
LLVM
后端

ARC实现原理

- llvm(Low Level Virtual Machine)底层虚拟机

它是一个编译器的基础建设，是为了任意一种编程语言写成的程序，利用虚拟技术，创造出编译时期，链结时期，运行时期以及“闲置时期”的优化

Xcode3之前，用的是GCC
Xcode3,GCC仍然保留，但是也推出了LLVM，苹果推荐
LLVM-GCC混合编译器，但还不是默认编译器
Xcode4,LLVM-GCC成为默认编译器，但GCC仍保留
Xcode4.2,LLVM3.0成为默认编译器,纯用GCC不复可能
Xcode4.6,LLVM升级到4.2版本
Xcode5,LLVM-GCC被遗弃，新的编译器是LLVM5.0，从GCC过渡到LLVM的时代正式完成

编译器代码优化帮我们加入了内存释放

```
int main()
{
    id a;
}
```

clang -S -fobjc-arc -emit-llvm main.m -o main.ll

```
void objc_storeStrong(id *object, id value) {
  id oldValue = *object;
  value = [value retain];
  *object = value;
  [oldValue release];
}
```

```
; ModuleID = 'main.m'
source_filename = "main.m"
target datalayout = "e-m:o-i64:64-
target triple = "x86_64-apple-maco

; Function Attrs: nounwind ssp uwt
define i32 @main() #0 {
  %1 = alloca i8*, align 8
  store i8* null, i8** %1, align 8
  call void @objc_storeStrong(i8** %1, i8* null) #1
  ret i32 0
}

declare void @objc_storeStrong(i8**, i8*)

attributes #0 = { nounwind ssp uwtable "correctly-rounded-divide-sqrt-fp-math="
false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-point
er-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-j
ump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16
,+fxsr,+mmx,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-
soft-float"="false" }
attributes #1 = { nounwind }
"main.ll" 28L, 1392C
```

IOS硬件

- 《一只IPhone的全球之旅》中介绍，iphone用了专用的芯片处理触摸屏的信息处理和手势识别，硬件的差异估计才是比较根本的。用cpu处理ui显然会卡。“著名电子设备调查机构 iSuppli 就曾经指出，Apple 打破业内常规的把最大部分成本花费在了改善用户体验上，因此，我们看 iPhone 的硬件架构就可以发现，为了处理一个小小的触控屏，apple动用了三块芯片，一个 Broadcom 的模拟信号处理器，用来处理触控屏传感器传来的模拟信号，转换为一组代表 x、y位置信息的数据流，一个飞利浦（NXP）ARM7CPU，用来作为手势算法处理器，把触控指令解析出来（后期 iPhone型号把这两个芯片整合在一起成为一个双核结构的处理器，称为改进型 Broadcom芯片），而主处理器则有一个高级别的优先独立线程专门处理触控操作类指令。
- 相对于其他电容触控手机大多把除了模数转换之外的大部分触控控制任务都交给主 CPU 的做法，iPhone 能够有“一触即发”的操作快感就不难理解了。

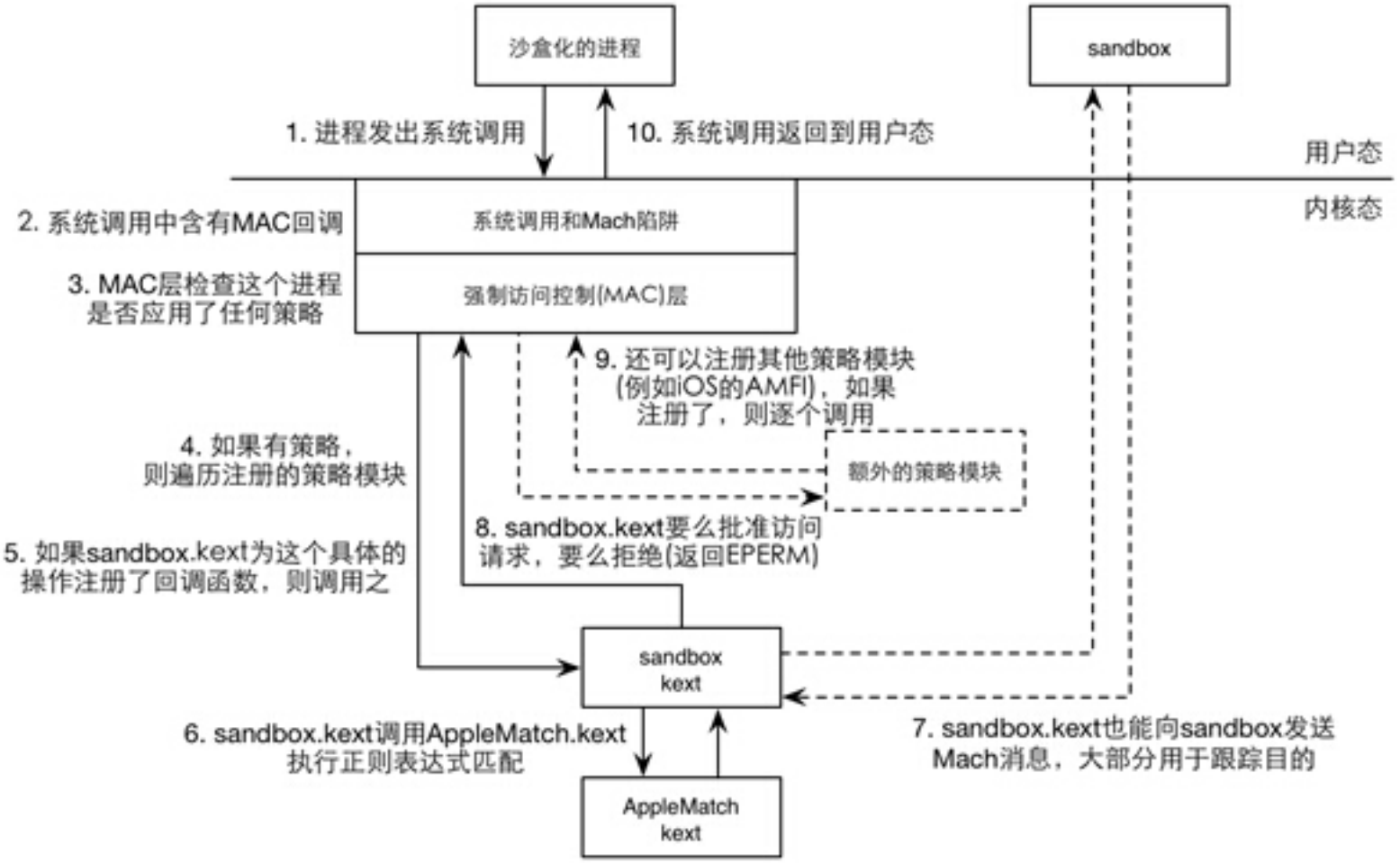
对比优缺点

- 优点：除了语言上的问题，伪后台，不同与android的垃圾回收机制，对用户体验的高度投入，使得ios需要较少的内存，较小的电池，后台占用cpu少，使得app更流畅
- 缺点：伪后台

（监狱）沙盒模式

- UID/GID permission
- iOS Mandatory Access Control

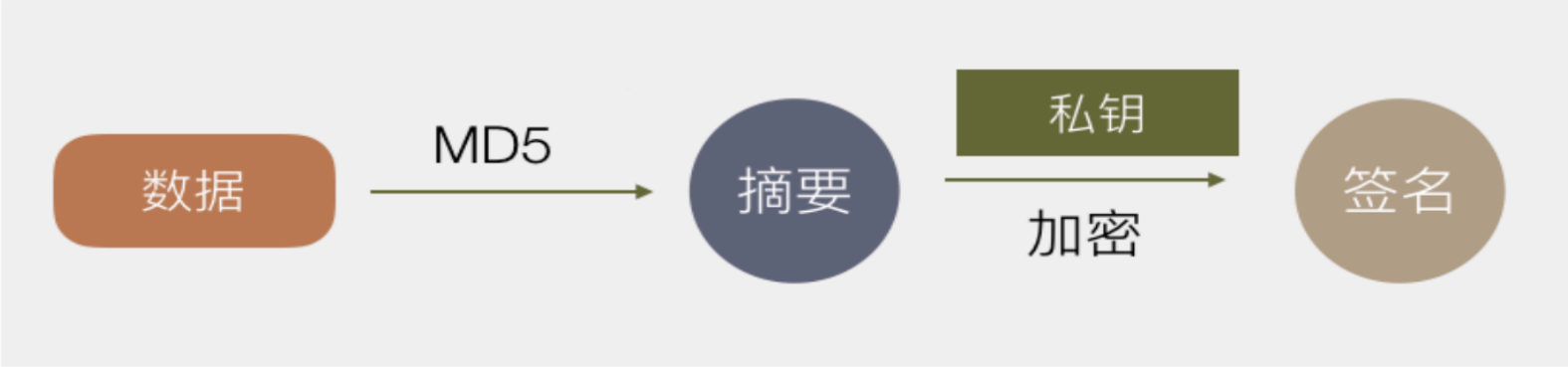
MAC

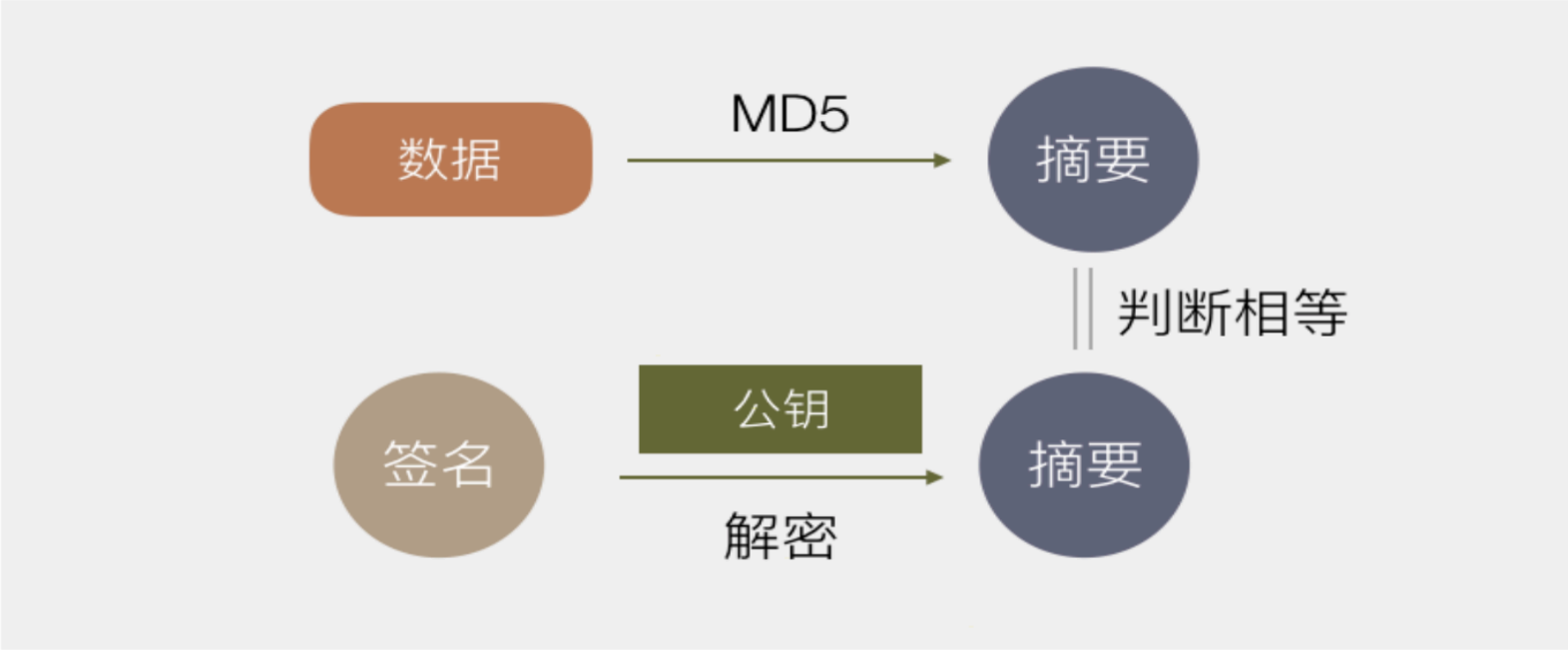

沙盒化的进程
sandbox
1. 进程发出系统调用
10. 系统调用返回到用户态
用户态
内核态
2. 系统调用中含有MAC回调
系统调用和Mach陷阱
3. MAC层检查这个进程
是否应用了任何策略
强制访问控制(MAC)层
9. 还可以注册其他策略模块
(例如iOS的AMFI)，如果
注册了，则逐个调用
4. 如果有策略，
则遍历注册的策略模块
额外的策略模块
8. sandbox.kext要么批准访问
请求，要么拒绝(返回EPERM)
5. 如果sandbox.kext为这个具体的
操作注册了回调函数，则调用之
sandbox
kext
6. sandbox.kext调用AppleMatch.kext
执行正则表达式匹配
7. sandbox.kext也能向sandbox发送
Mach消息，大部分用于跟踪目的
AppleMatch
kext

Kext(kernel extension)

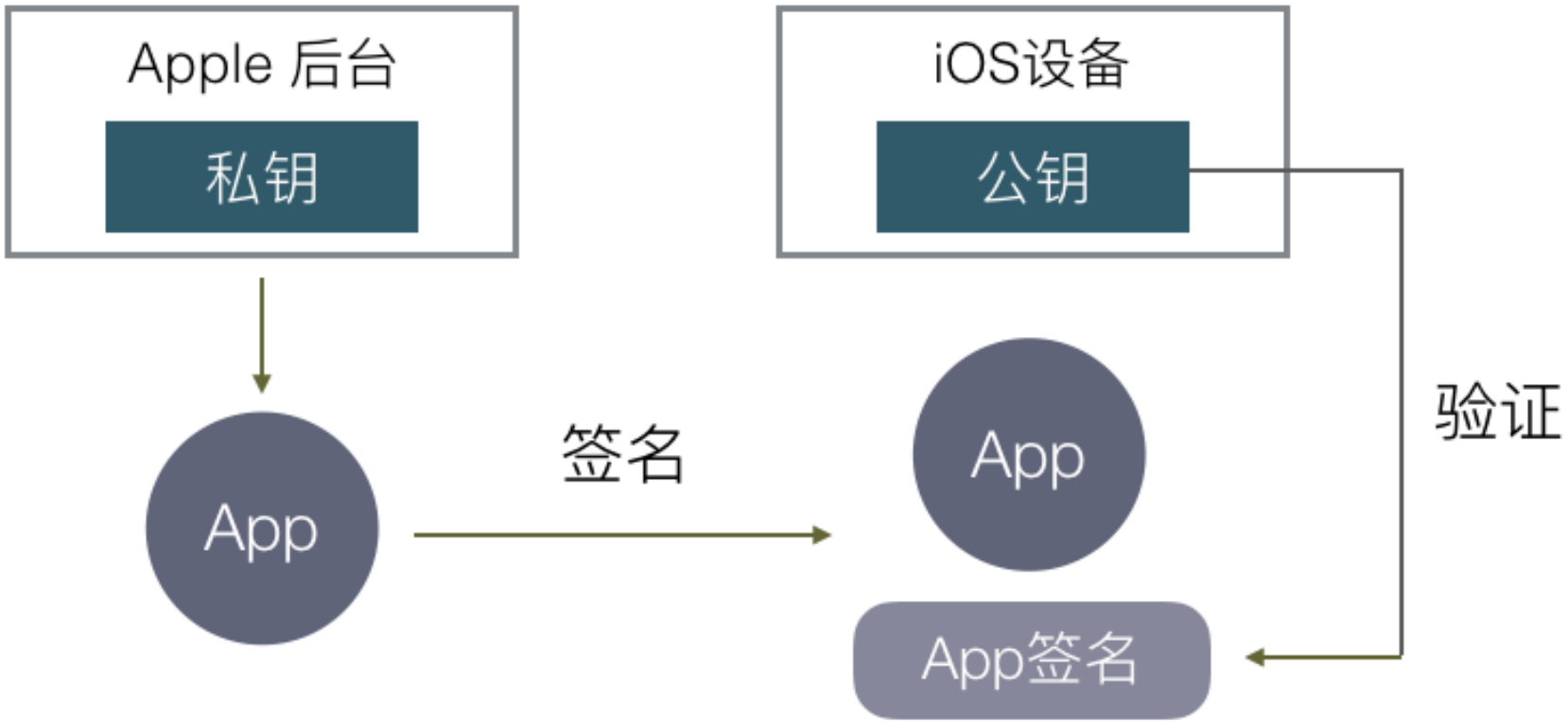
文件/子目录	内　　容
CodeDirectory	kext 的代码目录文件
CodeRequirements	kext 的代码需求设置
CodeResources	代码资源 XML 文件，包含 kext 中文件的散列和规则
CodeSignature	kext 的代码签名——通常包含苹果的数字证书
Info.plist	bundle 清单属性列表
MacOS	这个目录下包含了实际的 kext 二进制代码，二进制代码文件的类型为 BUNDLE(Mach-O 类型 8)或 KEXTBUNDLE(用于 64 位，Mach-O 类型 11)
_CodeSignature	包含 Code*文件的目录，这些文件实际上是上述 Code*文件的符号连接
version.plist	表示 kext 版本信息的属性列表

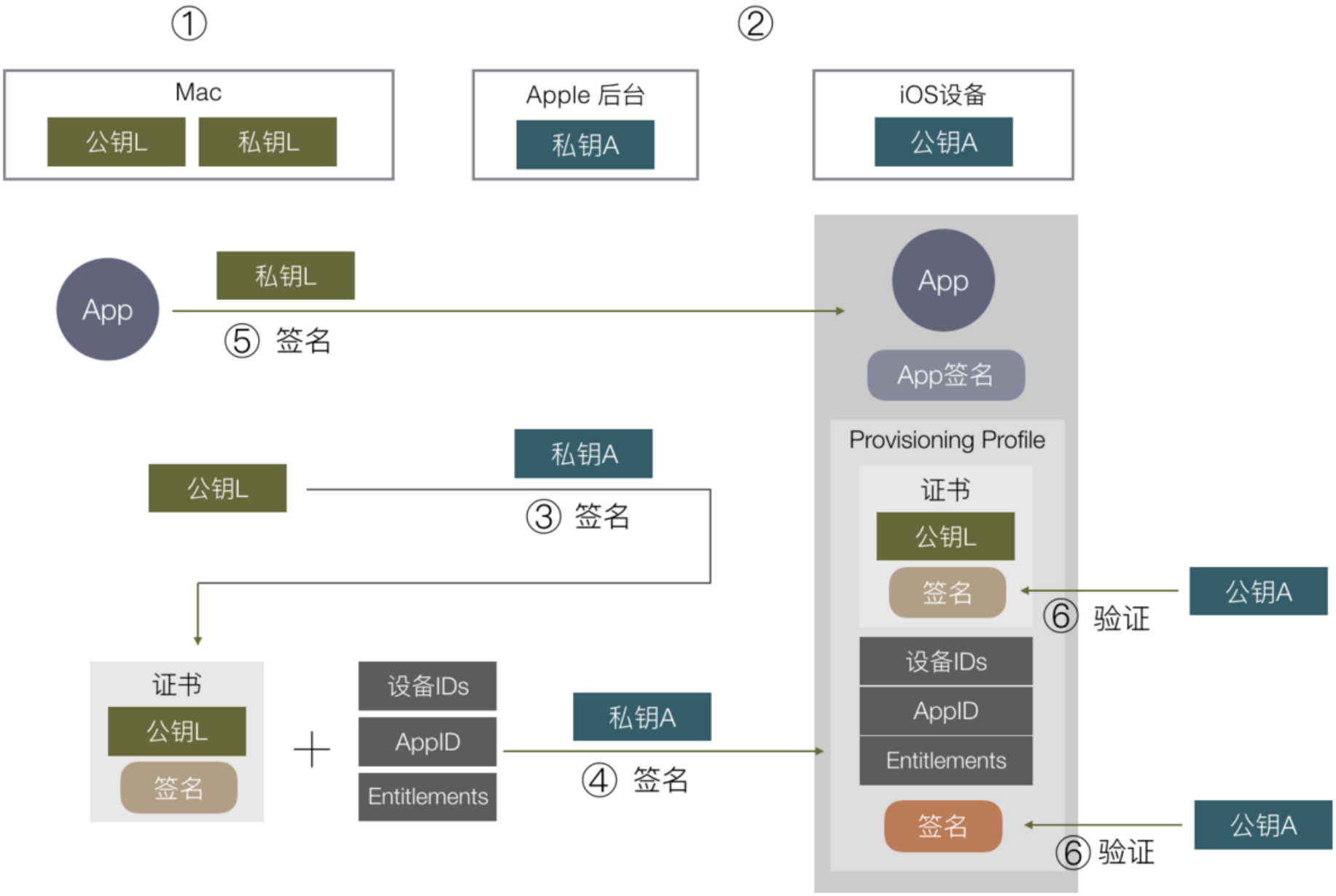
签名原理

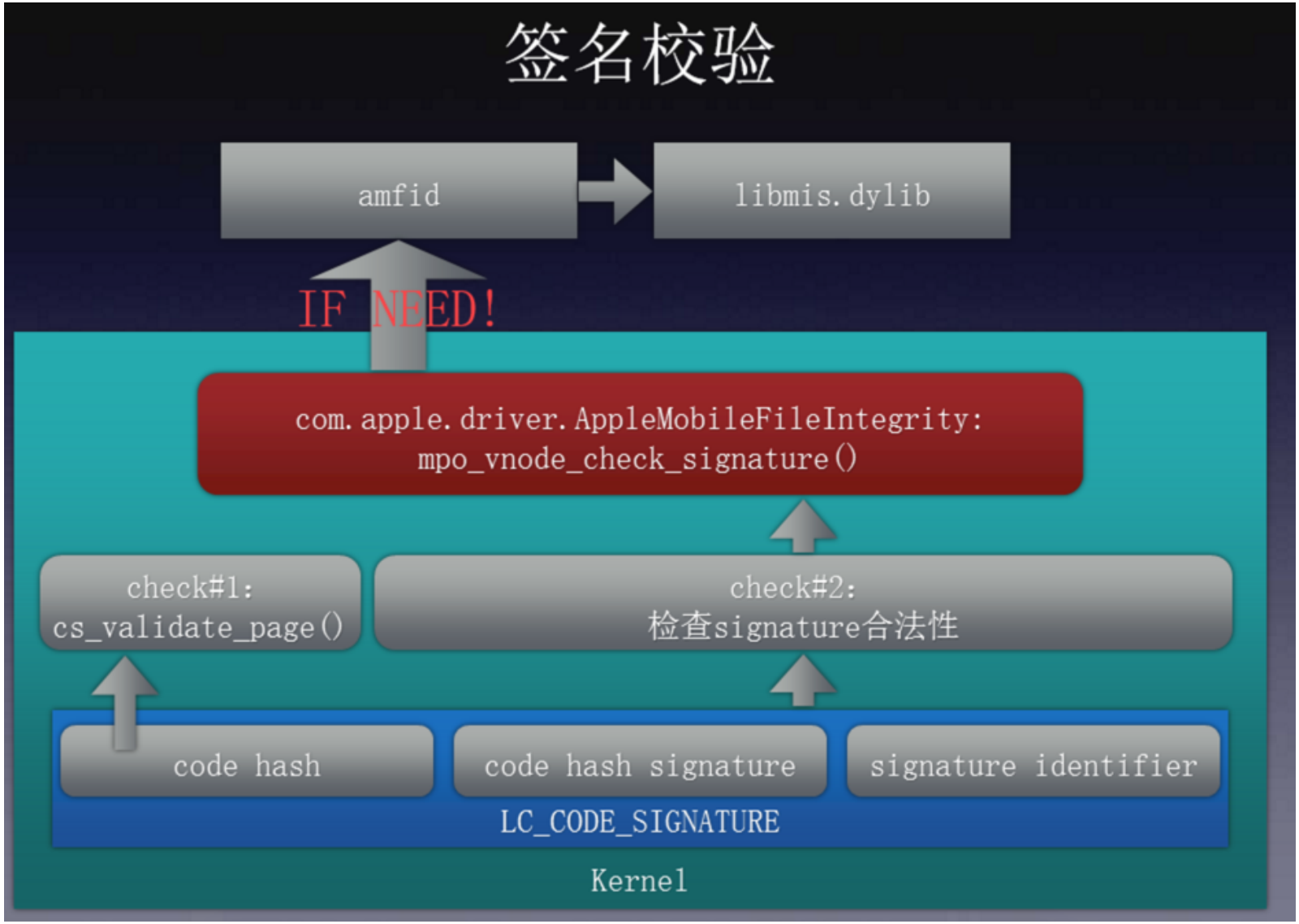




苹果商店签名




①
②
Mac
公钥L
私钥L
Apple 后台
私钥A
iOS设备
公钥A
App
私钥L
⑤ 签名
App
App签名
Provisioning Profile
证书
公钥L
签名
公钥A
⑥ 验证
私钥A
公钥L
③ 签名
证书
公钥L
签名
设备IDs
AppID
Entitlements
私钥A
④ 签名
设备IDs
AppID
Entitlements
签名
公钥A
⑥ 验证


签名校验
amfid
libmis.dylib
IF NEED!
com.apple.driver.AppleMobileFileIntegrity:
mpo_vnode_check_signature()
check#1:
cs_validate_page()
check#2:
检查signature合法性
code hash
code hash signature
signature identifier
LC_CODE_SIGNATURE
Kernel

总结

- 为什么快，内存使用少，耗电少？

伪后台（智能后台），APNs(推送)，代码中释放内存，没有系统垃圾回收模块。使得后台程序消耗很少的cpu和内存，硬件上对用户触摸做了较大的投入。

为什么安全？

沙盒使得app只能在自己的空间中处理，并限制了api方法调用，签名模式，使得只有从appstroe下载的app（企业发布，需要手动认证证书），才能在硬件上运行。