另外我还分享一些整理的Android开发相关的学习资料（100G左右），资料包括开发工具、入门基础知识、进阶、项目实战的源码及视频，还有电子书。在我公众号“ **优派编程** ”后台回复“ **57966** ”获取！

介绍

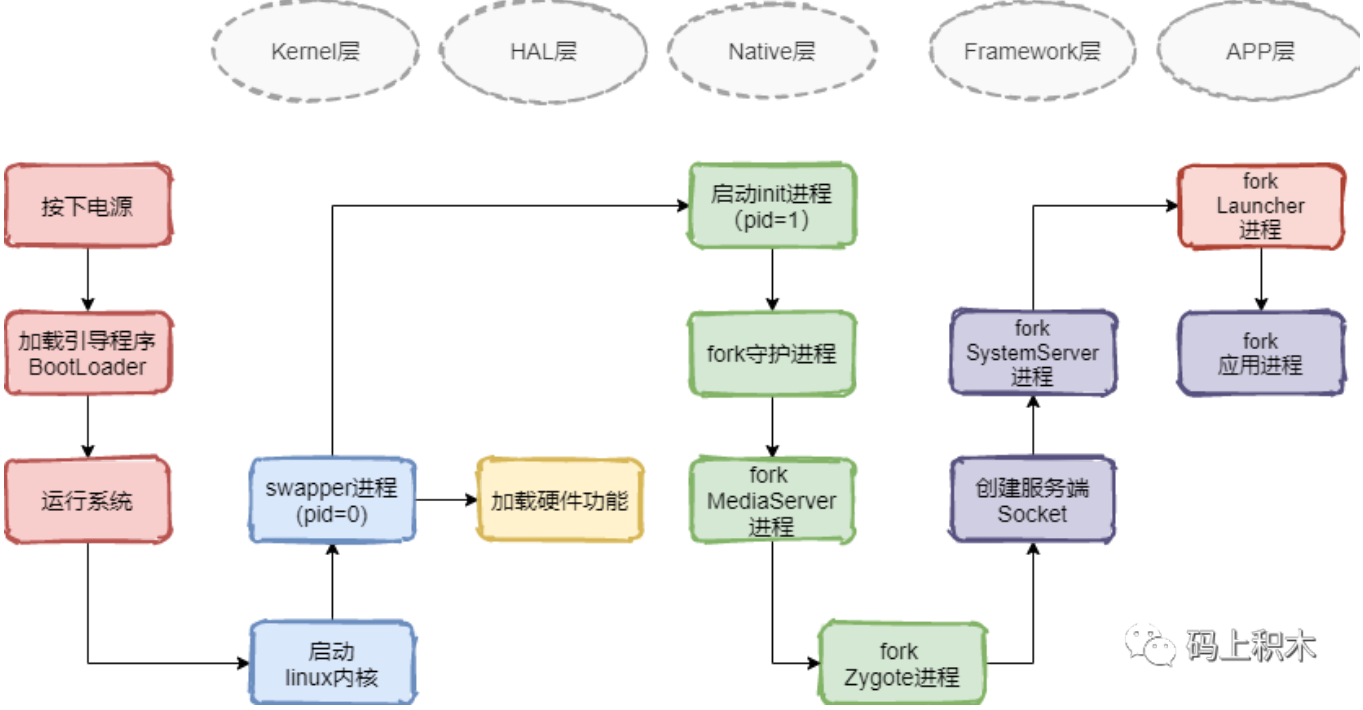
《面试题思考与解答》系列期刊是将每月的知识点进行总结汇总。

要声明的一点是：面试题的目的不是为了让大家背题，而是从不同维度帮助大家复习，取长补短。

希望大家都能找到满意的工作。

以下为2022年1月刊内容。

简述Android系统启动流程



Android系统中启动的第一个进程是哪个？

(一般不会问这么深，可以作为知识扩展了解)

这个问题涉及到内核层的启动情况了。

在Kernel层，Android系统会启动linux内核。

我们知道Android的核心系统服务都是基于Linux内核的，但是这个Linux内核到底该怎么理解呢？

Linux内核并不指的是Linux操作系统，内核只包括最基本的内存模型，进程调度，权限安全等等。操作系统值得是一个更广的概念，不光有内核，还有自己的设备驱动，应用程序框架以及一些应用程序软件等等。所以Android、Ubuntu等都是基于Linux内核的不同的操作系统。

所以启动了linux内核，就是启动了内核中内存模型，进程调度，安全机制，加载驱动等等，而linux内核中的功能都需要上册的虚拟机进行调用执行。

内核中就启动了系统中的第一个进程：

- swapper进程(pid=0)，该进程又称为idle进程, 系统初始化过程Kernel由无到有开创的第一个进程, 用于初始化进程管理、内存管理。并且会加载屏幕硬件，相机硬件等，这一步就会涉及到待会说到的HAL层了。

第一个用户级进程是哪个？

init进程是Android系统中用户空间的第一个进程，是所有用户进程的鼻祖。

启动入口在`system/core/init/init.cpp`文件中，init进程中主要做了这些事：

- **孵化出用户守护进程。**守护进程就是运行在后台的特殊进程，它不存在控制终端，会周期性处理一些任务。比如logd进程，就是用来进行日志的读写操作。
- **启动了一些重要服务。**比如开机动画。
- **孵化了Zygote进程。**Zygote进程大家都或多或少了解一些了，我们所有的应用程序都是由它孵化出来的。
- **孵化了Media Server进程，用来启动和管理整个C++ framework**，比如相机服务（camera Service）。

Zygote进程做了些什么工作？

- **创建服务端Socket**，为后续创建进程通信做准备。
- **加载虚拟机。**没错，在Zygote进程中，会去加载下层的虚拟机。
- **fork了System Server进程。**`SystemServer`进程大家应该都熟悉了吧，是Zygote fork的第一个进程，负责启动和管理Java Framework层，包括`ActivityManagerService`，`PackageManagerService`，`WindowManagerService`、`binder`线程池等等。这就涉及到APP的启动流程了，后续几篇会细说下。
- **fork了第一个应用进程——Launcher**，以及后续的一些系统应用进程，这就到了最上面一层——应用层了。

Activity启动流程中，大部分都是用Binder通讯，为啥跟Zygote通信的时候要用socket呢

此题来自每日一问（*https://www.wanandroid.com/wenda/show/10482*）
评论区主要有以下观点：

- `ServiceManager`不能保证在zygote起来的时候已经初始化好，所以无法使用Binder。
- `Socket`的所有者是 root，只有系统权限用户才能读写，多了安全保障。
- `Binder`工作依赖于多线程，但是fork的时候是不允许存在多线程的，多线程情况下进程fork容易造成死锁，所以就不用Binder了。

反射可以修改final类型成员变量吗？

`final`我们应该都知道，修饰变量的时候代表是一个常量，不可修改。那利用反射能不能达到修改的效果呢？

我们先试着修改一个用`final`修饰的`String`变量。

```
public class User {
  private final String name = "Bob";
  private final Student student = new Student();

  public String getName() {
    return name;
  }

  public Student getStudent() {
```

```
    return student;
  }
}


  User user = new User();
  Class clz = User.class;
  Field field1 = null;
  try{
    field1=clz.getDeclaredField("name");
    field1.setAccessible(true);
    field1.set(user,"xixi");
    System.out.println(user.getName());
  }catch(NoSuchFieldException e){
    e.printStackTrace();
  }catch(IllegalAccessException e){
    e.printStackTrace();
  }
```

打印出来的结果，还是Bob，也就是没有修改到。

我们再修改下`student`变量试试：

```
field1 = clz.getDeclaredField("student");
field1.setAccessible(true);
field1.set(user, new Student());

打印：
修改前com.example.studynote.reflection.Student@77459877
修改后com.example.studynote.reflection.Student@72ea2f77
```

可以看到，对于正常的对象变量即使被final修饰也是可以通过反射进行修改的。

这是为什么呢？为什么`String`不能被修改，而普通的对象变量可以被修改呢？

先说结论，其实`String`值也被修改了，只是我们无法通过这个对象获取到修改后的值。

这就涉及到JVM的内联优化了：

> 内联函数，编译器将指定的函数体插入并取代每一处调用该函数的地方（上下文），从而节省了每次调用函数带来的额外时间开支。

简单的说，就是JVM在处理代码的时候会帮我们优化代码逻辑，比如上述的`final`变量，已知`final`修饰后不会被修改，所以获取这个变量的时候就直接帮你在编译阶段就给赋值了。

所以上述的`getName`方法经过JVM编译内联优化后会变成：

```
public String getName() {
  return "Bob";
}
```

所以无论怎么修改，都获取不到修改后的值。

有的朋友可能提出直接获取`name`呢？比如这样：

```
//修改为public
public final String name = "Bob";

//反射修改后，打印user.name
field1=clz.getDeclaredField("name");
field1.setAccessible(true);
field1.set(user,"xixi");
System.out.println(user.name);
```

不好意思，还是打印出来Bob。这是因为`System.out.println(user.name)`这一句在经过编译后，会被写成：

```
System.out.println(user.name)

//经过内联优化

System.out.println("Bob")
```

所以：

反射是可以修改final变量的，但是如果是基本数据类型或者`String`类型的时候，无法通过对象获取修改后的值，因为JVM对其进行了内联优化。

那有没有办法获取修改后的值呢？

有，可以通过反射中的`Field.get(Object obj)`方法获取：

```
//获取field对应的变量在user对象中的值
System.out.println("修改后"+field.get(user));
```

反射获取static静态变量

说完了`final`，再说说`static`，怎么修改`static`修饰的变量呢？

我们知道，静态变量是在类的实例化之前就进行了初始化（类的初始化阶段），所以静态变量是跟着类本身走的，跟具体的对象无关，所以我们获取变量就不需要传入对象，直接传入`null`即可：

```
public class User {
 public static String name;
}


field2 = clz.getDeclaredField("name");
field2.setAccessible(true);
//获取静态变量
Object getname=field2.get(null);
System.out.println("修改前"+getname);

//修改静态变量
```

```
field2.set(null, "xixi");
System.out.println("修改后"+User.name);
```

如上述代码：

- `Field.get(null)` 可以获取静态变量。
- `Field.set(null,object)` 可以修改静态变量。

怎么提升反射效率

1、缓存重复用到的对象

利用缓存，其实我不说大家也都知道，在平时项目中用到多次的对象也会进行缓存，谁也不会多次去创建。

但是，这一点在反射中尤为重要，比如`Class.forName`方法，我们做个测试：

```
long startTime = System.currentTimeMillis();
Class clz = Class.forName("com.example.studynote.reflection.User");
User user;
int i = 0;
while (i < 1000000) {
  i++;
  //方法1，直接实例化
  user = new User();
  //方法2，每次都通过反射获取class，然后实例化
  user = (User) Class.forName("com.example.studynote.reflection.User").newInstance();
  //方法3，通过之前反射得到的class进行实例化
  user = (User) clz.newInstance();
}

System.out.println("耗时：" + (System.currentTimeMillis() - startTime));
```

打印结果：

```
1、直接实例化
耗时：15

2、每次都通过反射获取class，然后实例化
耗时：671

3、通过之前反射得到的class进行实例化
耗时：31
```

所以看出来，只要我们合理的运用这些反射方法，比如`Class.forName`，`Constructor`，`Method`，`Field`等，尽量在循环外就缓存好实例，就能提高反射的效率，减少耗时。

2、setAccessible(true)

之前我们说过当遇到私有变量和方法的时候，会用到`setAccessible(true)`方法关闭安全检查。这个安全检查其实也是耗时的。

所以我们在反射的过程中可以尽量调用`setAccessible(true)`来关闭安全检查，无论是否是私有的，这样也能提高反射的效率。

3、ReflectASM

> ReflectASM 是一个非常小的 Java 类库，通过代码生成来提供高性能的反射处理，自动为 get/set 字段提供访问类，访问类使用字节码操作而不是 Java 的反射技术，因此非常快

> ASM是一个通用的Java字节码操作和分析框架。它可以用于修改现有类或直接以二进制形式动态生成类。

简单的说，这是一个类似反射，但是不同于反射的高性能库。他的原理是通过ASM库，生成了一个新的类，然后相当于直接调用新的类方法，从而完成反射的功能。

感兴趣的可以去看看源码，实现原理比较简单——
https://github.com/EsotericSoftware/reflectasm。

小总结：经过上述三种方法，我想反射也不会那么可怕到大大影响性能的程度了，如果真的发现反射影响了性能以及实际使用的情况，也许可以研究下，是否是因为没用对反射和没有处理好反射相关的缓存呢？

反射原理

如果我们试着查看这些反射方法的源码，会发现最终都会走到`native`方法中，比如`getDeclaredField`方法会走到

```
public native Field getDeclaredField(String name) throws NoSuchFieldException;
```

那么在底层，是怎么获取到类的相关信息的呢？

首先回顾下JVM加载Java文件的过程：

- 编译阶段，`.java`文件会被编译成`.class`文件，`.class`文件是一种二进制文件，内容是JVM能够识别的机器码。
- `.class`文件里面依次存储着类文件的各种信息，比如：版本号、类的名字、字段的描述和描述符、方法名称和描述、是不是`public`、类索引、字段表集合，方法集合等等数据。
- 然后，JVM中的类加载器会读取字节码文件，取出二进制数据，加载到内存中，并且解析`.class`文件的信息。
- 类加载器会获取类的二进制字节流，在内存中生成代表这个类的`java.lang.Class`对象。
- 最后会开始类的生命周期，比如连接、初始化等等。

而反射，就是去操作这个`java.lang.Class`对象，这个对象中有整个类的结构，包括属性方法等等。

总结来说就是，.class是一种有顺序的结构文件,而Class对象就是对这种文件的一种表示，所以我们能从Class对象中获取关于类的所有信息，这就是反射的原理。

在java有Serializable的前提下，Android为什么设计出了Parcelable？

java中的序列化方式Serializable效率比较低，主要有以下原因：

- Serializable在序列化过程中会创建大量的临时变量，这样就会造成大量的GC。
- Serializable使用了大量反射，而反射操作耗时。
- Serializable使用了大量的IO操作，也影响了耗时。

所以Android就像重新设计了IPC方式Binder一样，重新设计了一种序列化方式，结合Binder的方式，对上述三点进行了优化，一定程度上提高了序列化和反序列化的效率。

Serializable、Parcelable、Json等序列化方式我们该怎么选择？

先说说序列化的用处，主要用在三个方面：

1、内存数据传输

内存传输方面，主要用Parcelable。一是因为Parcelable在内存传输的效率比Serializable高。二是因为在Android中很多传输数据的方法中，自带了对于Serializable、Parcelable类型的传输方法。比如:

- Bundle.putParcelable,
- Intent putExtra(String name, Parcelable value)

等等吧，基本上对象传输的方法都支持了，所以这也是Parcelable的优势。

2、 数据持久化（本地存储）

如果只针对Serializable和Parcelable两种序列化方式，需要选择Serializable。

首先，Serializable本身就是存储到二进制文件，所以用于持久化比较方便。而Parcelable序列化是在内存中操作，如果进程关闭或者重启的时候，内存中的数据就会消失，那么Parcelable序列化用来持久化就有可能会失败，也就是数据不会连续完整。

而且Parcelable还有一个问题是兼容性，每个Android版本可能内部实现都不一样，知识用于内存中也就是传递数据的话是不影响的，但是如果持久化可能就会有问题了，低版本的数据拿到高版本可能会出现兼容性问题。

但是实际情况，对于Android中的对象本地化存储，一般是以数据库、SP的方式进行保存。

3、 网络传输

而对于网络传输的情况，一般就是使用JSON了。主要有以下几点原因：

1. 轻量级，没有多余的数据。
2. 与语言无关，所以能兼容所有平台语言。
3. 易读性，易解析。

Parcelable一定比Serializable快吗？

正常情况下，对象在内存中进行传输确实是`Parcelable`比较快，但是`Serializable`是有缓存的概念的，有人做了一个比较有趣的实验：

当序列化一个超级大的对象图表（表示通过一个对象，拥有通过某路径能访问到其他很多的对象），并且每个对象有10个以上属性时，并且`Serializable`实现了`writeObject()`以及`readObject()`，在平均每台安卓设备上，`Serializable`序列化速度大于`Parcelable` 3.6倍，反序列化速度大于1.6倍。

具体原因就是因为`Serilazable`的实现方式中，是有缓存的概念的，当一个对象被解析过后，将会缓存在`HandleTable`中，当下一次解析到同一种类型的对象后，便可以向二进制流中，写入对应的缓存索引即可。但是对于Parcel来说，没有这种概念，每一次的序列化都是独立的，每一个对象，都当作一种新的对象以及新的类型的方式来处理。

具体过程可以看看这篇：*https://juejin.cn/post/6854573218334769166*

为什么Java提供了Serializable的序列化方式，而不是直接使用json或者xml？

我觉得是历史遗留问题。

有的人可能会想到各种理由，比如可以标记哪些类可以被序列化。又或者可以通过UID来标示反序列化为同一个对象。等等。

但是我觉得最大的问题还是历史遗留问题，在以前，json还没有成为大家认同的数据结构，所以Java就设计出了`Serializable`的序列化方式来解决对象持久化和对象传输的问题。然后Java中各种API就会依赖于这种序列化方式，这么些年过去了，Java体系的庞大也造成难以改变这个问题，牵一发而动全身。

为什么我这么说呢？

主要有两点依据：

1. 曾经Oracle Java平台组的架构师说过，删除Java的序列化机制并且提供给用户可以选择的序列化方式（比如json）是他们计划中的一部分，因为Java序列化也造成了很多Java漏洞。具体可以参见文章：*https://www.infoworld.com/article/3275924/oracle-plans-to-dump-risky-java-serialization.html*

2. 因为在`Serializable`类的介绍注释中，明确说到推荐大家选择JSON 和 GSON库，因为它简洁、易读、高效。

```
* <h3>Recommended Alternatives</h3>
* <strong>JSON</strong> is concise, human-readable and efficient. Android
* includes both a {@link android.util.JsonReader streaming API} and a {@link
* org.json.JSONObject tree API} to read and write JSON. Use a binding library
* like <a href="http://code.google.com/p/google-gson/">GSON</a> to read and
* write Java objects directly.
```

Window是什么

窗口。你可以理解为手机上的整个画面，所有的视图都是通过Window呈现的，比如`Activity`、`dialog`都是附加在Window上的。Window类的唯一实现是`PhoneWindow`，这个名字就更加好记了吧，手机窗口呗。

那Window到底在哪里呢？我们看到的View是Window吗？是也不是。

如果说的只是Window概念的话，那可以说是的，View就是Window的存在形式，Window管理着View。

如果说是Window类的话，那确实不是View，唯一实现类`PhoneWindow`管理着当前界面上的View，包括根布局——`DecorView`，和其他子view的添加删除等等。

不知道你晕没有，我总结下，Window是个概念性的东西，你看不到他，如果你能感知它的存在，那么就是通过View，所以View是Window的存在形式，有了View，你才感知到View外层有一个皇帝的新衣——window。

WindowManager是什么？和WMS的关系？

`WindowManager`就是用来管理Window的，实现类为`WindowManagerImpl`，实际工作会委托给`WindowManagerGlobal`类中完成。

而具体的Window操作，WM会通过Binder告诉WMS，WMS做最后的真正操作Window的工作，会为这个Window分配Surface，并绘制到屏幕上。

怎么添加一个Window？

```
var windowParams: WindowManager.LayoutParams = WindowManager.LayoutParams()
windowParams.flags = WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
windowParams.type = WindowManager.LayoutParams.TYPE_SYSTEM_DIALOG
var btn = Button(this)
windowManager.addView(btn, windowParams)
```

简单贴了下代码，加了一个Button。

有的朋友可能会疑惑了，这明明是个Button，是个View啊，咋成了Window？

刚才说过了，View是Window的表现形式，在实际实现中，添加window其实就是添加了一个你看不到的window，并且里面有View才能让你感觉得到这个是一个Window。

所以通过`windowManager`添加的View其实就是添加Window的过程。

这其中还有两个比较重要的属性：`flags`和`type`，下面会依次说到。

Window怎样可以显示到锁屏界面

Window的`flag`可以控制Window的显示特性，也就是该怎么显示、`touch`事件处理、与设备的关系、等等。所以这里问的锁屏界面显示也是其中的一种`Flag`。

```
// Window不需要获取焦点，也不接受各种输入事件。
public static final int FLAG_NOT_FOCUSABLE = 0x00000008;

// @deprecated Use {@link android.R.attr#showWhenLocked} or
// {@link android.app.Activity#setShowWhenLocked(boolean)} instead to prevent an
// unintentional double life-cycle event.


// 窗口可以在锁屏的 Window 之上显示
public static final int FLAG_SHOW_WHEN_LOCKED = 0x00080000;
```

Window三种类型都存在的情况下，显示层级是怎样。

`Type`表示Window的类型，一共三种：

- 应用Window。对应着一个`Activity`，Window层级为1~99，在视图最下层。
- 子Window。不能单独存在，需要附属在特定的父Window之中(如`Dialog`就是子Window)，Window层级为1000~1999。
- 系统Window。需要声明权限才能创建的Window，比如`Toast`和系统状态栏，Window层级为2000-2999，处在视图最上层。

可以看到，区别就是有个Window层级（z-ordered），层级高的能覆盖住层级低的，离用户更近。

Window就是指PhoneWindow吗？

如果有人问我这个问题，我肯定心里要大大的疑惑了🤔。

可不就是`PhoneWindow`吗？都唯一实现类了，净问些奇怪问题。

但是面试的时候遇到这种问题总要答啊？这时候就要扯出Window的概念了。

如果指的Window类，那么PhoneWindow作为唯一实现类，一般指的就是PhoneWindow。

如果指的Window这个概念，那肯定不是指PhoneWindow，而是存在于界面上真实的View。当然也不是所有的View都是Window，而是通过WindowManager添加到屏幕的view才是Window，所以PopupWindow是Window，上述问题中添加的单个View也是Window。

PhoneWindow什么时候被创建的？

熟悉Activity启动流程的朋友应该知道，启动过程会执行到ActivityThread的handleLaunchActivity方法，这里初始化了WindowManagerGlobal，也就是WindowManager实际操作Window的类，待会会看到：

```
public Activity handleLaunchActivity(ActivityClientRecord r,
                    PendingTransactionActions pendingActions, Intent customIntent) {
  //...
  WindowManagerGlobal.initialize();
  //...
  final Activity a = performLaunchActivity(r, customIntent);
  //...
  return a;
}
```

然后会执行到performLaunchActivity中创建Activity，并调用attach方法进行一些数据的初始化(伪代码)：

```
final void attach() {
  //初始化PhoneWindow
  mWindow = new PhoneWindow(this, window, activityConfigCallback);
  mWindow.setWindowControllerCallback(mWindowControllerCallback);
  mWindow.setCallback(this);

  //和WindowManager关联
  mWindow.setWindowManager(
       (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
       mToken, mComponent.flattenToString(),
       (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);

  mWindowManager = mWindow.getWindowManager();
}
```

可以看到，在Activity的attach方法中，创建了PhoneWindow，并且设置了callback，windowManager。

这里的callback待会会说到，跟事件分发有关系，可以说是当前Activity和PhoneWindow建立联系。

要实现可以拖动的View该怎么做？

还是接着刚才的btn例子，如果要修改btn的位置，使用updateViewLayout即可，然后在ontouch方法中传入移动的坐标即可。

```
btn.setOnTouchListener { v, event ->
```

```
  val index = event.findPointerIndex(0)
  when (event.action) {
    ACTION_MOVE -> {
      windowParams.x = event.getRawX(index).toInt()
      windowParams.y = event.getRawY(index).toInt()
      windowManager.updateViewLayout(btn, windowParams)
    }
    else -> {
    }
  }
  false

}
```

Window的添加、删除和更新过程。

Window的操作都是通过`WindowManager`来完成的，而`WindowManager`是一个接口，他的实现类是`WindowManagerImpl`，并且全部交给`WindowManagerGlobal`来处理。下面具体说下`addView`，`updateViewLayout`，和`removeView`。

1. addView

```
//WindowManagerGlobal.java
public void addView(View view, ViewGroup.LayoutParams params,
      Display display, Window parentWindow) {

    if (parentWindow != null) {
      parentWindow.adjustLayoutParamsForSubWindow(wparams);
    }

      ViewRootImpl root;
      View panelParentView = null;

      root = new ViewRootImpl(view.getContext(), display);
      view.setLayoutParams(wparams);

      mViews.add(view);
      mRoots.add(root);
      mParams.add(wparams);

      try {
        root.setView(view, wparams, panelParentView);
      }
    }
  }
```

- 这里可以看到，创建了一个`ViewRootImpl`实例，这样就说明了每个Window都对应着一个`ViewRootImpl`。
- 然后通过add方法修改了`WindowManagerGlobal`中的一些参数，比如mViews—存储了所有Window所对应的View，mRoots——所有Window所对应的`ViewRootImpl`，mParams—所有Window对应的布局参数。
- 最后调用了`ViewRootImpl`的`setView`方法,继续看看。

```
final IWindowSession mWindowSession;

mWindowSession = WindowManagerGlobal.getWindowSession();

public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
  //
  requestLayout();
```

```
    res = mWindowSession.addToDisplay(mWindow,);
}
```

setView方法主要完成了两件事，一是通过requestLayout方法完成异步刷新界面的请求，进行完整的view绘制流程。其次，会通过IWindowSession进行一次IPC调用，交给到WMS来实现Window的添加。

其中mWindowSession是一个Binder对象，相当于在客户端的代理类，对应的服务端的实现为Session，而Session就是运行在SystemServer进程中，具体就是处于WMS服务中，最终就会调用到这个Session的addToDisplay方法，从方法名就可以猜到这个方法就是具体添加Window到屏幕的逻辑，具体就不分析了，下次说到屏幕绘制的时候再细谈。

2. updateViewLayout

```
public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
//...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;

    view.setLayoutParams(wparams);

    synchronized (mLock) {
        int index = findViewLocked(view, true);
        ViewRootImpl root = mRoots.get(index);
        mParams.remove(index);
        mParams.add(index, wparams);
        root.setLayoutParams(wparams, false);
    }
}
```

这里更新了WindowManager.LayoutParams和ViewRootImpl.LayoutParams，然后在ViewRootImpl内部同样会重新对View进行绘制，最后通过IPC通信，调用到WMS的relayoutWindow完成更新。

3. removeView

```
public void removeView(View view, boolean immediate) {
    if (view == null) {
        throw new IllegalArgumentException("view must not be null");
    }

    synchronized (mLock) {
        int index = findViewLocked(view, true);
        View curView = mRoots.get(index).getView();
        removeViewLocked(index, immediate);
        if (curView == view) {
            return;
        }

        throw new IllegalStateException("Calling with view " + view
                + " but the ViewAncestor is attached to " + curView);
    }
}


private void removeViewLocked(int index, boolean immediate) {
    ViewRootImpl root = mRoots.get(index);
    View view = root.getView();

    if (view != null) {
        InputMethodManager imm = view.getContext().getSystemService(InputMethodManager.class);
        if (imm != null) {
```

```
        imm.windowDismissed(mViews.get(index).getWindowToken());
      }
    }
    boolean deferred = root.die(immediate);
    if (view != null) {
      view.assignParent(null);
      if (deferred) {
        mDyingViews.add(view);
      }
    }
}
```

该方法中，通过view找到mRoots中的对应索引，然后同样走到`ViewRootImpl`中进行View删除工作，通过`die`方法，最终走到`dispatchDetachedFromWindow()`方法中，主要做了以下几件事：

- 回调`onDetachedFromeWindow`。
- 垃圾回收相关操作。
- 通过`Session`的`remove()`在WMS中删除Window。
- 通过`Choreographer`移除监听器。

Activity、PhoneWindow、DecorView、ViewRootImpl 的关系？

看完上面的流程，我们再来理理这四个小伙伴之间的关系：

- `PhoneWindow` 其实是 Window 的唯一子类，是 Activity 和 View 交互系统的中间层，用来管理View的，并且在Window创建（添加）的时候就新建了`ViewRootImpl`实例。
- `DecorView` 是整个 View 层级的最顶层，`ViewRootImpl`是`DecorView` 的parent，但是他并不是一个真正的 View，只是继承了`ViewParent`接口，用来掌管View的各种事件，包括`requestLayout`、`invalidate`、`dispatchInputEvent` 等等。

Window中的token是什么，有什么用？

token？又是个啥呢？刚才window操作过程中也没出现啊。

token其实大家应该工作中会发现一点踪迹，比如application的上下文去创建`dialog`的时候，就会报错：

```
unable to add window --token null
```

所以这个token跟window操作是有关系的，翻到刚才的`addview`方法中，还有个细节我们没说到，就是`adjustLayoutParamsForSubWindow`方法。

```
//Window.java
void adjustLayoutParamsForSubWindow(WindowManager.LayoutParams wp) {
  if (wp.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
      wp.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
    //子Window
    if (wp.token == null) {
```

```
        View decor = peekDecorView();
        if (decor != null) {
          wp.token = decor.getWindowToken();
        }
      }
    } else if (wp.type >= WindowManager.LayoutParams.FIRST_SYSTEM_WINDOW &&
        wp.type <= WindowManager.LayoutParams.LAST_SYSTEM_WINDOW) {
      //系统Window
    } else {
      //应用Window
      if (wp.token == null) {
        wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;
      }

    }
}
```

上述代码分别代表了三个Window的类型：

- 子Window。需要从`decorview`中拿到token。
- 系统Window。不需要token。
- 应用Window。直接拿mAppToken，mAppToken是在`setWindowManager`方法中传进来的，也就是新建Window的时候就带进来了token。

然后在WMS中的`addWindow`方法会验证这个token，下次说到WMS的时候再看看。

所以这个token就是用来验证是否能够添加Window，可以理解为权限验证，其实也就是为了防止开发者乱用context创建window。

拥有token的context（比如Activity）就可以操作Window。没有token的上下文（比如Application）就不允许直接添加Window到屏幕（除了系统Window）。

Application中可以直接弹出Dialog吗？

这个问题其实跟上述问题相关：

- 如果直接使用Application的上下文是不能创建Window的，而Dialog的Window等级属于子Window，必须依附与其他的父Window，所以必须传入Activity这种有window的上下文。
- 那有没有其他办法可以在Application中弹出`dialog`呢？有，改成系统级Window：

//检查权限

```
if (!Settings.canDrawOverlays(this)) {
  val intent = Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION)
  intent.data = Uri.parse("package:$packageName")
  startActivityForResult(intent, 0)
}

dialog.window.setType(WindowManager.LayoutParams.TYPE_SYSTEM_DIALOG)

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

- 另外还有一种办法，在Application类中，可以通过registerActivityLifecycleCallbacks监听Activity生命周期，不过这种办法也是传入了Activity的context，只不过在Application类中完成这个工作。

关于事件分发，事件到底是先到DecorView还是先到Window的？

经过上述一系列问题，是不是对Window印象又深了点呢？最后再看一个问题，这个是wanandroid论坛上看到的(*https://wanandroid.com/wenda/show/12119*)，

这里的window可以理解为PhoneWindow，其实这道题就是问事件分发在Activity、DecorView、PhoneWindow中的顺序。

当屏幕被触摸，首先会通过硬件产生触摸事件传入内核，然后走到FrameWork层（具体流程感兴趣的可以看看参考链接），最后经过一系列事件处理到达ViewRootImpl的processPointerEvent方法，接下来就是我们要分析的内容了：

```
//ViewRootImpl.java
 private int processPointerEvent(QueuedInputEvent q) {
       final MotionEvent event = (MotionEvent)q.mEvent;
       ...
       //mView分发Touch事件，mView就是DecorView
       boolean handled = mView.dispatchPointerEvent(event);
       ...
     }

//DecorView.java
   public final boolean dispatchPointerEvent(MotionEvent event) {
     if (event.isTouchEvent()) {
       //分发Touch事件
       return dispatchTouchEvent(event);
     } else {
       return dispatchGenericMotionEvent(event);
     }
   }

   @Override
   public boolean dispatchTouchEvent(MotionEvent ev) {
     //cb其实就是对应的Activity
     final Window.Callback cb = mWindow.getCallback();
     return cb != null && !mWindow.isDestroyed() && mFeatureId < 0
         ? cb.dispatchTouchEvent(ev) : super.dispatchTouchEvent(ev);
   }


//Activity.java
   public boolean dispatchTouchEvent(MotionEvent ev) {
     if (ev.getAction() == MotionEvent.ACTION_DOWN) {
       onUserInteraction();
     }
     if (getWindow().superDispatchTouchEvent(ev)) {
       return true;
     }
     return onTouchEvent(ev);
   }

//PhoneWindow.java
   @Override
   public boolean superDispatchTouchEvent(MotionEvent event) {
     return mDecor.superDispatchTouchEvent(event);
   }

//DecorView.java
```

```
//DecorView.java
  public boolean superDispatchTouchEvent(MotionEvent event) {
    return super.dispatchTouchEvent(event);
  }
```

事件的分发流程就比较清楚了：

`ViewRootImpl`——>`DecorView`——>`Activity`——>`PhoneWindow`——>`DecorView`——>`ViewGroup`

（这其中就用到了`getCallback`参数，也就是之前`addView`中传入的`callback`，也就是`Activity`本身）

但是这个流程确实有些奇怪，为什么绕来绕去的呢，光`DecorView`就走了两遍。

参考链接中的说法我还是比较认同的，主要原因就是解耦。

- `ViewRootImpl`并不知道有`Activity`这种东西存在，它只是持有了`DecorView`。所以先传给了`DecorView`，而`DecorView`知道有AC，所以传给了AC。
- `Activity`也不知道有`DecorView`，它只是持有`PhoneWindow`，所以这么一段调用链就形成了。

怎么理解Binder？

在java层面，其实`Binder`就是一个实现了`IBinder`接口的类。

真正跨进程的部分还是在客户端发起远程调用请求之后，系统底层封装好，交给服务端的时候。而这个系统底层封装，其实就是发生在Linux内核中。

而在内核中完成这个通信关键功能的还是`Binder`，这次不是`Binder`类了，而是`Binder`驱动。

> 驱动你可以理解为一种硬件接口，可以帮助操作系统来控制硬件设备。

`Binder`驱动被添加运行到Linux内核空间，这样，两个不同进程就可以通过访问内核空间来完成数据交换：把数据传给`Binder`驱动，然后处理好再交给对方进程,完成跨进程通信。

而刚才通过AIDL的例子我们可以知道，客户端在请求服务端通信的时候，并不是直接和服务端的某个对象联系，而是用到了服务端的一个代理对象，通过对这个代理对象操作，然后代理类会把方法对应的code、传输的序列化数据、需要返回的序列化数据交给底层，也就是`Binder`驱动。

然后`Binder`驱动把对应的数据交给服务器端，等结果计算好之后，再由Binder驱动把数据返回给客户端。

最后借用《Android开发艺术探索》书中的内容总结下，希望大家回味回味。

```
直观的说，Binder是一个类，实现了IBinder接口。
```

```
从IPC（进程间通信）角度来说，Binder是Android中一种跨进程通信方式。
还可以理解为一种虚拟的物理设备，它的设备驱动是/dev/binder。
从Android FrameWork角度来说，Binder是ServiceManager连接各种Manager（ActivityManager，WindowManager等等）和响应Manager
从Android应用层来说，Binder是客户端和服务端进行通信的媒介。
```

怎么理解ServiceManager

ServiceManager其实是为了管理系统服务而设置的一种机制，每个服务注册在ServiceManager中，由ServiceManager统一管理，我们可以通过服务名在ServiceManager中查询对应的服务代理，从而完成调用系统服务的功能。所以ServiceManager有点类似于DNS，可以把服务名称和具体的服务记录在案，供客户端来查找。

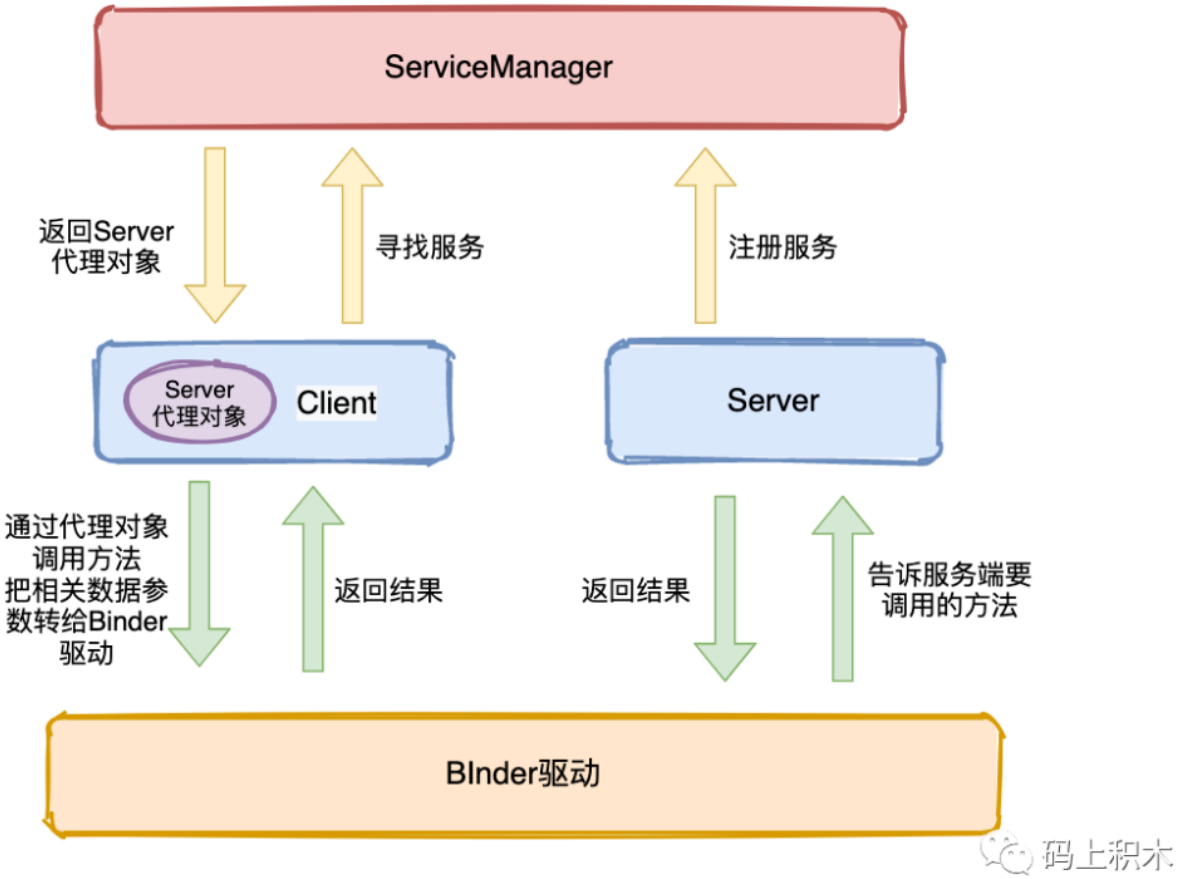
在我们这个AIDL的案例中，能直接获取到服务端的Service，也就直接能获取到服务端的代理类IMsgManager，所以就无需通过ServiceManager这一层来寻找服务了。

而且ServiceManager本身也运行在一个单独的线程，所以它本身也是一个服务端，客户端其实是先通过跨进程获取到ServiceManager的代理对象，然后通过ServiceManager代理对象再去找到对应的服务。

而ServiceManager就像我们刚才AIDL中的Service一样，是可以直接找到的，他的句柄永远是0，是一个"众所周知"的句柄，所以每个APP程序都可以通过binder机制在自己的进程空间中创建一个ServiceManager代理对象。

所以通过ServiceManager查找系统服务并调用方法的过程是进行了两次跨进程通信。

APP进程——>ServiceManager进程——>系统服务进程（比如ActivityManagerService）



网络通信的过程，以及中间用了什么协议

这个问题我之前专门做了一个动画，大家可以翻到上一篇文章看看：

网络数据原来是这么传输的（结合动画解析）

https://mp.weixin.qq.com/s/PFhA3WdS-2aSdbWqGyTETQ

再简单总结下：

客户端：

1. 在浏览器输入网址。
2. 浏览器解析网址，并生成http请求消息。
3. 浏览器调用系统解析器，发送消息到DNS服务器查询域名对应的ip。
4. 拿到ip后，和请求消息一起交给操作系统协议栈的TCP模块。
5. 将数据分成一个个数据包，并加上TCP报头形成TCP数据包。
6. TCP报头包括发送方端口号、接收方端口号、数据包的序号、ACK号。
7. 然后将TCP消息交给IP模块。

8. IP模块会添加IP头部和MAC头部。

9. IP头部包括IP地址，为IP模块使用，MAC头部包括MAC地址，为数据链路层使用。

10. IP模块会把整个消息包交给网络硬件，也就是数据链路层，比如以太网，WIFI等。

11. 然后网卡会将这些包转换成电信号或者在光信号，通过网线或者光纤发送出去，再由路由器等转发设备送达接收方。

服务器端：

1. 数据包到达服务器的数据链路层，比如以太网，然后会将其转换为数据包（数字信号）交给IP模块。

2. IP模块会将MAC头部和IP头部后面的内容，也就是TCP数据包发送给TCP模块。

   TCP模块会解析TCP头信息，然后和客户端沟通表示收到这个数据包了。

3. TCP模块在收到消息的所有数据包之后，就会封装好消息，生成相应报文发给应用层，也就是HTTP层。

4. HTTP层收到消息，比如是HTML数据，就会解析这个HTML数据，最终绘制到浏览器页面上。

TCP连接过程，三次握手和四次挥手，为什么？

连接阶段（三次握手）：

- 创建套接字Socket，服务器会在启动的时候就创建好，客户端是在需要访问服务器的时候创建套接字。
- 然后发起连接操作，其实就是Socket的connect方法。
- 这时候客户端会生成一个TCP数据包。这个数据包的TCP头部有几个重要信息：SYN、ACK、Seq、Ack。

> SYN，同步序列编号，是TCP/IP建立连接时使用的握手信号，如果这个值为1就代表是连接消息。ACK，确认标志，如果这个值为1就代表是确认消息。Seq，数据包序号，是发送数据的一个顺序编号。Ack Number，确认数字号，是接收数据的一个顺序编号。

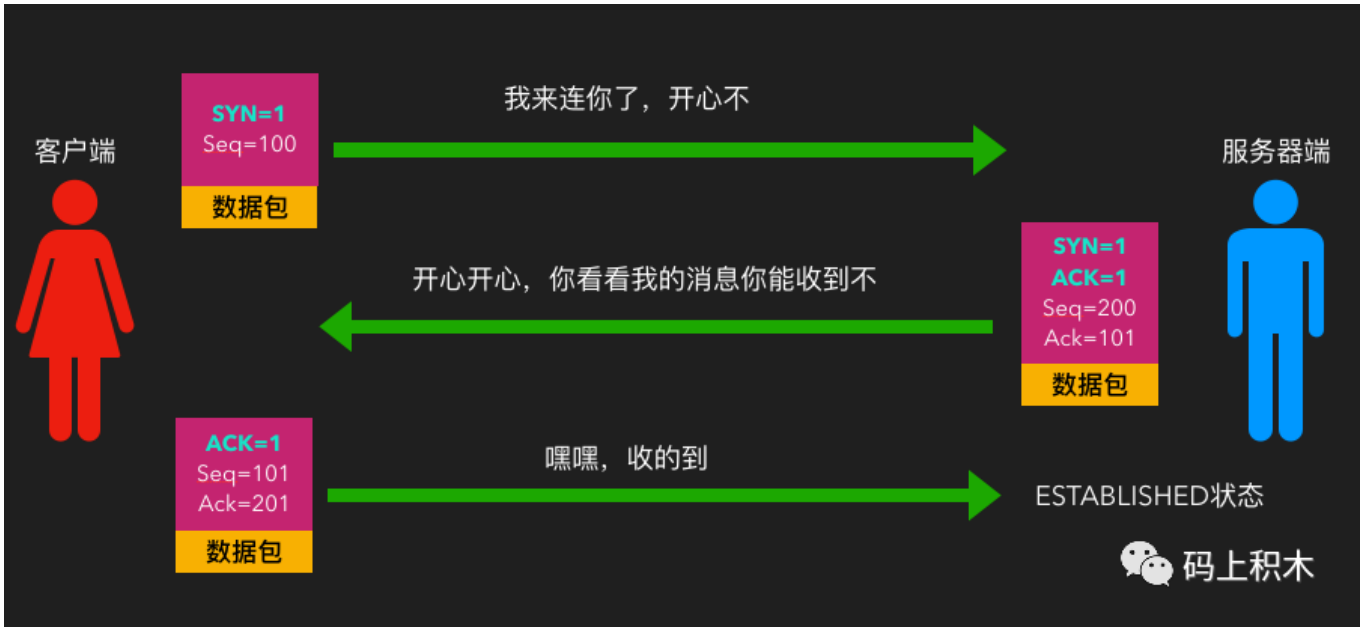
- 所以客户端就生成了这样一个数据包，其中头部信息的控制位SYN设置为1，代表连接。SEQ设置一个随机数，代表初始序号，比如100。
- 然后服务器端收到这个消息，知道了客户端是要来连接的（SYN=1），知道了传输数据的初始序号（SEQ=100）。
- 服务器端也要生成一个数据包发送给客户端，这个数据包的TCP头部会包含：表示我也要连接你的SYN（SYN=1），我已经收到了你的上个数据包的确认号ACK=1（Ack=Seq+1=101），以及服务器端随机生成的一个序号Seq（比如Seq=200）。
- 最后客户端收到这个消息后，表示客户端到服务器的连接是无误了，然后再发送一个数据包表示也确认收到了服务器发来的数据包，这个数据包的头部就主要就是一个ACK=1（Ack=Seq+1=201）。
- 至此，连接成功，三次握手结束，后面数据就会正常传输，并且每次都要带上TCP头部中的Seq和Ack。

这里有个问题是关于为什么需要三次握手？

最主要的原因就是需要通信双方都确认自己的消息被准确传达过去了。

A发送消息给B，B回一条消息表示我收到了，这个过程就保证了A的通信能力。B发送消息给A，A回一条消息表示我收到了，这个过程就保证了B的通信能力。

也就是四条消息能保证双方的消息发送都是正常的，其中B回消息和B发消息，可以融合为一次消息，所以就有了三次握手。



数据传输阶段：

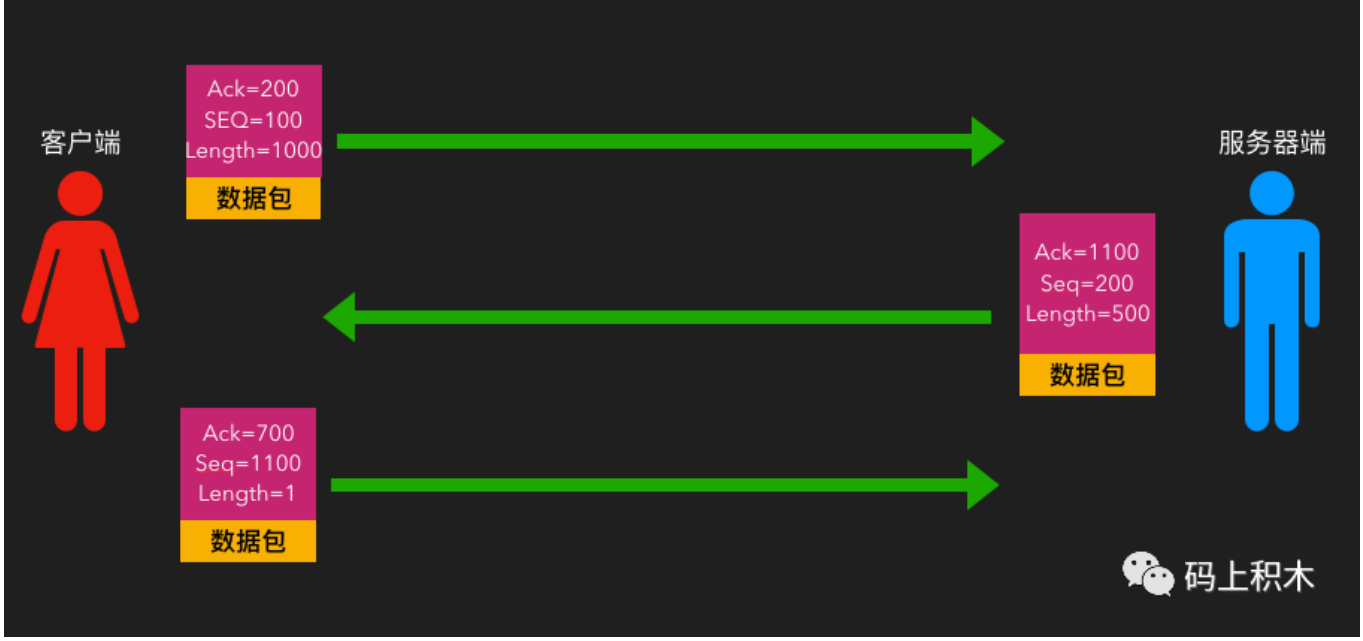
数据传输阶段有个改变就是Ack确认号不再是Seq+1了，而是Seq+数据长度。例如：

- A发送给B的数据包（Seq=100，长度=1000字节）
- B回给A的数据包（Ack=100+1000=1100）

这就是一次数据传输的头部信息，Ack代表下个数据包应该从哪个字节开始所以等于上个数据包的Seq+长度，Seq就等于上个数据包的Ack。

当然，TCP通信是双向的，所以实际数据每个消息都会有Seq和Ack：

- A发送给B的数据包（Ack=200，Seq=100，长度=1000字节）
- B回给A的数据包（Ack=100+1000=1100，Seq=上一个数据包的Ack=200，长度=500字节）
- A发送给B数据包（Seq=1100，Ack=200+500=700）



断开阶段（四次挥手）：

和连接阶段一样，TCP头部也有一个专门用作关闭连接的值叫做FIN。

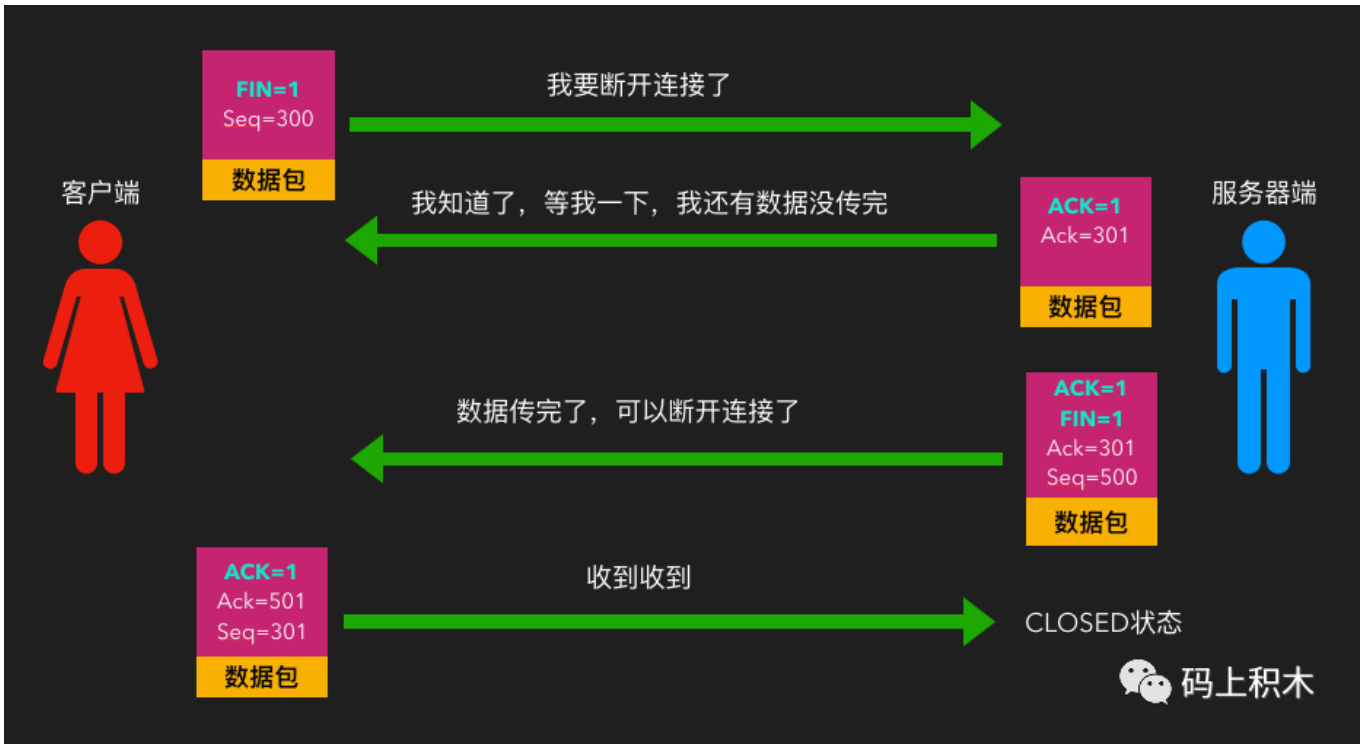
- 客户端准备关闭连接，会发送一个TCP数据包，头部信息中包括（FIN=1代表要断开连接）。
- 服务器端收到消息，回复一个数据包给客户端，头部信息中包括Ack确认号。但是此时服务器端的正常业务可能没有完成，还要处理下数据，收个尾。
- 客户端收到消息。
- 服务器继续处理数据。
- 服务器处理数据完毕，准备关闭连接，会发送一个TCP数据包给客户端，头部信息中包括（FIN=1代表要断开连接）。
- 客户端端收到消息，回复一个数据包给服务器端，头部信息中包括Ack确认号。
- 服务器收到消息，到此服务器端完成连接关闭工作。
- 客户端经过一段时间（2MSL），自动进入关闭状态，到此客户端完成连接关闭工作。

> MSL 是 Maximum Segment Lifetime，报文最大生存时间，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

这里有个问题是关于为什么需要四次挥手？

A发送断开消息给B，B回一条消息表示我收到了，这个过程就保证了A断开成功。B发送断开消息给A，A回一条消息表示我收到了，这个过程就保证了B断开成功。

其实和连接阶段的区别就在于，这里的B的确认消息和断开消息不能融合。因为A要断开的时候，B可能还有数据要处理要发送，所以要等正常业务处理完，在发送断开消息。



常用的状态码

- 1XX - 临时消息。服务器收到请求，需要请求者继续操作。
- 2XX - 请求成功。请求成功收到，理解并处理。
- 3XX - 重定向。需要进一步的操作以完成请求。
- 4XX - 客户端错误。请求包含语法错误或无法完成请求。
- 5XX - 服务器错误。服务器在处理请求的过程中发生了错误。

常见状态码：

200 OK - 客户端请求成功301 - 资源(网页等)被永久转移到其它URL302 - 临时跳转400 Bad Request - 客户端请求有语法错误，不能被服务器所理解404 - 请求资源不存在，错误的URL。500 - 服务器内部发生了不可预期的错误。503 Server Unavailable - 服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

讲一下TCP协议和UDP协议的区别和场景

我先说两个场景，大家可能就比较能理解了。

1. 第一个场景，浏览网页。（TCP场景）

- 我们访问网页，网页肯定要把所有数据都正确的显示出来吧，如果这个过程中丢包了，那么肯定也会重新传包，不可能只显示一部分网页。（保证数据正确性）

- 同样，网页中的内容肯定也需要是顺序的。比如我一个抽奖，不可能还没抽就把奖给你了。（保证数据的顺序）

- 再来，在这个对数据要求严格的过程中，我们肯定需要两方建立起一个可靠的连接，也就是我们上述说到的要经过三次握手才开始传输数据，并且每次发数据包都需要回执。（面向连接的）

- 而这种连接中传输数据就是用的字节流，也就是有根管道，你想怎么传数据都行，想怎么接受数据也都可以，只要在这一根管道里面。

所以这种需要数据准确、顺序不能错、要求稳定可靠的场景就需要用到TCP。

2. 第二个场景，打游戏。（UDP场景）

打游戏最最重要的就是即时，不然我这个技能发出去了你那边还没被打中，这就玩不了了。

- 所以UDP是需要保证数据的即时性，而不保证每个数据包都正确接收到，即使丢包了，也不会去找丢的那个是什么包，因为要显示当前时间的当前数据包。（不保证数据正确性和数据顺序，可能会丢包）

- 同样，为了数据的即时性，UDP也就不会去建立连接了，不需要什么三次握手，每次你还要确认收没收到。管你收没收到，我只要快速把每个数据包丢给你就行了。（面向无连接的）

- 因为是无连接的，所以就不需要用到字节流，直接每次丢一个数据报给你，接收方也只能接受一个数据报（不能和其他发送方的数据报混淆）。（基于数据报的）

如果你还是有点晕，可以看看这篇文章（亚当和夏娃），很形象的比喻：

https://www.zhihu.com/question/51388497?sort=created

socket和WebSocket

虽然这两个货名字类似，但其实不是一个层级的概念。

- socket，套接字。上文说过了，在TCP建立连接的过程中，是调用了Socket的相关API，建立了这个连接通道。所以它只是一个接口，一个类。

- WebSocket，是和HTTP同等级，属于应用层协议。它是为了解决长时间通信的问题，由HTML5规范引出，是一种建立在TCP协议基础上的全双工通信的协议，同样下层也需要TCP建立连接,所以也需要socket。

> 科普：WebSocket在TCP连接建立后，还要通过Http进行一次握手，也就是通过Http发送一条GET请求消息给服务器，告诉服务器我要建立WebSocket连接了，你准备好哦，具体做法就是在头部信息中添加相关参数。然后服务器响应我知道了，并且将连接协议改成WebSocket，开始建立长连接。

如果硬要说这两者有关系，那就是WebSocket协议也用到了TCP连接，而TCP连接用到了Socket的API。

Https的连接建立过程

说完了HTTP和TCP/IP，再说说HTTPS。

上一篇文章说了HTTPS是怎么保证数据安全传输
https://mp.weixin.qq.com/s/dbmwBVxHkvQ0fzWaSdtPYg

其中主要就是用到了数字证书。

现在完整看看Https连接建立（也叫TLS握手流程）：

1、客户端发送 Client Hello 数据包消息。

这个消息内容包括一个随机数（randomC）,加密族（密钥交换算法也就是非对称加密算法、对称加密算法、哈希算法）,Session ID(用作恢复回话)。

客户端要建立通信，在TCP握手之后，会发送第一个消息，也叫Client Hello消息。这个消息主要发了以上的一些内容，其中密文族就是把客户端这边支持的一些算法发给服务器，然后服务器拿来和服务器支持的算法一比较，就能得出双方都支持的最优算法了。

2、服务器回复三个数据包消息：Server Hello，Certificate，Server Hello Done。

Server Hello消息内容包括一个随机数（randomS），比较后得出的加密族，Session ID(用作恢复回话)。

到现在，双方已经有两个随机数了，待会再看看这两个随机数是干嘛的。然后加密算法刚才说过了，服务器协商出了三种算法并发回给客户端。

Certificate消息就是发送数字证书了。这里就不细说了。

Server Hello Done消息就是个结束标志，表示已经把该发的消息都发给你了。

3、对称密钥生成过程

1）首先，客户端会对发来的证书进行验证，比如数字签名、证书链、证书有效期、证书状态。

2）证书校验完毕后，然后客户端会用证书里的服务器公钥加密发送一个随机数 pre—master secret ，服务器收到之后用自己的私钥解密。

3）到此，客户端和服务器就都有三个随机数了：randomC、randomS、pre—master secret。

4）然后客户端和服务器端分别按照固定的算法，用三个随机数生成对称密钥。

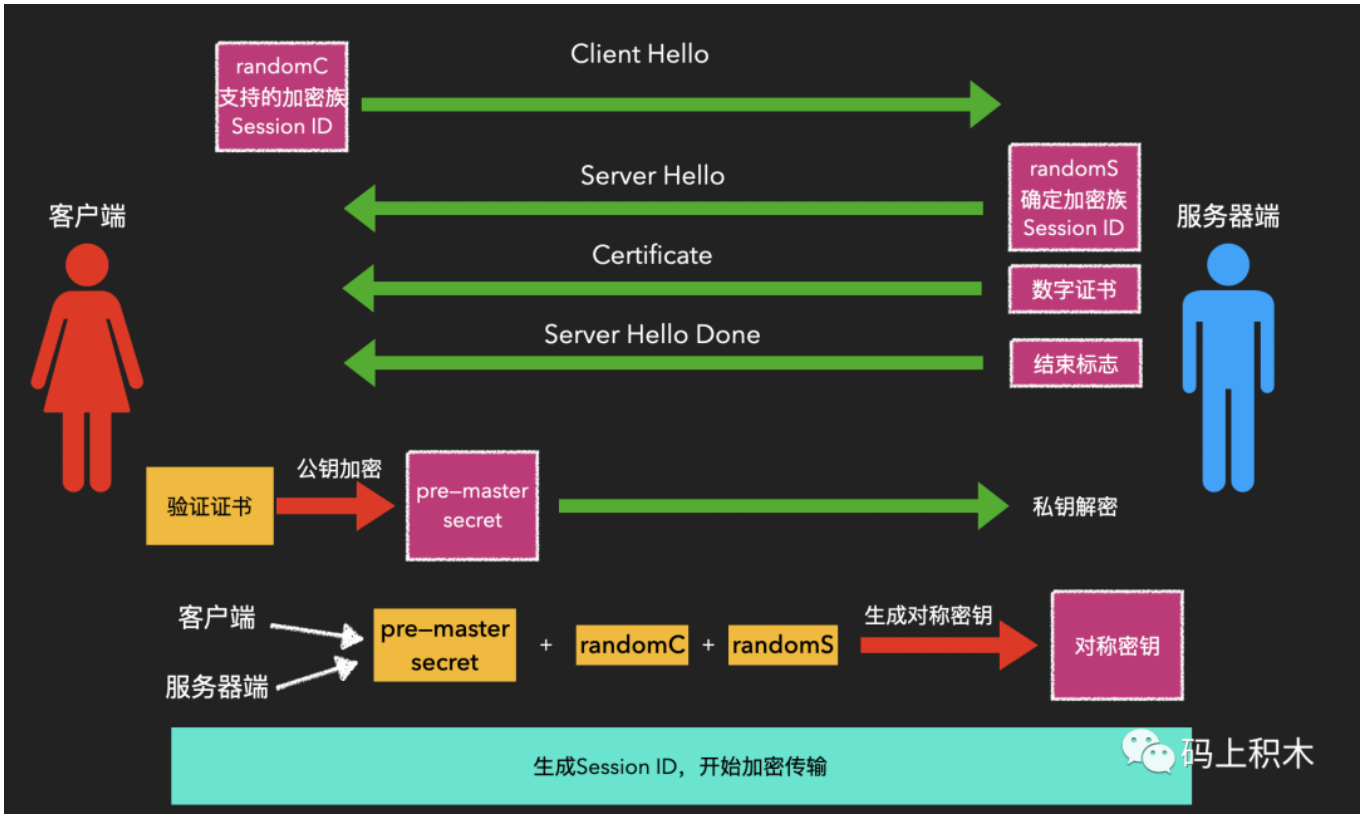
4、生成Session ID

这一步和开始两个hello消息中的Session ID对应起来了。

会生成会话的id，如果后续会话断开了，那么通过这个Session ID就可以恢复对话，不需要重新进行发送证书、生成密钥过程了。

5、用对称密钥传输数据

拿到对称密钥后，双方就可以使用对称密钥加密解密数据，进行正常通信了。



扩展：为什么要使用非对称加密算法协商出对称加密这种方法？

首先，网络传输数据对传输的速度要求比较高，在保证安全的前提下，所以采用了对称加密的方法，而不用耗时较多的非对称加密算法。

其次，在确定对称加密传输数据的前提下，如果传输对称加密的密钥是个涉及到安全的问题，所以就采用了安全性更高的非对称加密算法，加上证书链机制，保证了传输对称密钥相关数据的安全性。

请给我讲解一下数字签名，为什么真实可靠

数字签名，也就是上文中说的电子签名，再简单回顾下：

数字签名，其实也是一种非对称加密的用法。

它的使用方法是：

A使用私钥对数据的哈希值进行加密，这个加密后的密文就叫做签名，然后将这个密文和数据本身传输给B。

B拿到后，签名用公钥解密出来，然后和传过来数据的哈希值做比较，如果一样，就说明这个签名确实是A签的，而且只有A才可以签，因为只有A有私钥。

反应实际情况就是：

服务器端将数据，也就是我们要传的数据（公钥），用另外的私钥签名数据的哈希值，然后和数据（公钥）一起传过去。然后客户端用另外的公钥对签名解密，如果解密数据和数据（公钥）的哈希值一致，就能证明来源正确，不是被伪造的。

- 来源可靠。数字签名只能拥有私钥的一方才能签名，所以它的存在就保证了这个数据的来源是正确的
- 数据可靠。hash值是固定的，如果签名解密的数据和本身的数据哈希值一致，说明数据是未被修改的。

证书链安全机制

> 证书颁发机构（CA, Certificate Authority）即颁发数字证书的机构。是负责发放和管理数字证书的权威机构，并作为电子商务交易中受信任的第三方，承担公钥体系中公钥的合法性检验的责任。

实际情况中，服务器会拿自己的公钥以及服务器的一些信息传给CA，然后CA会返回给服务器一个数字证书，这个证书里面包括：

- 服务器的公钥
- 签名算法
- 服务器的信息，包括主机名等。
- CA自己的私钥对这个证书的签名

然后服务器将这个证书在连接阶段传给客户端，客户端怎么验证呢？

细心的小伙伴肯定知道，每个客户端，不管是电脑、手机都有自带的系统根证书，其中就会包括服务器数字证书的签发机构。所以系统的根证书会用他们的公钥帮我们对数字证书的签名进行解密，然后和证书里面的数据哈希值进行对比，如果一样，则代表来源是正确的,数据是没有被修改的。

当然中间人也是可以通过CA申请证书的，但是证书中会有服务器的主机名，这个主机名（域名、IP）就可以验证你的来源是来源自哪个主机。

扩展一下：

其实在服务器证书和根证书中间还有一层结构：叫中级证书，我们可以任意点开一个网页，点击左上角的🔒按钮就可以看到证书详情：

可以看到一般完整的SSL/TLS证书有三层结构：

- 第一层：根证书。也就是客户端自带的那些，根证书都是自签名，即用自己的公钥和私钥完成了签名的制作和验证。
- 第二层：中级证书。一般根证书是不会直接颁发服务器证书的，因为这种行为比较危险，如果发现错误颁发就很麻烦，需要涉及到跟证书的修改。所以一般会引用中间证书，根证书对中间证书进行签名，然后中间证书再对服务器证书进行签名，一层套一层。
- 第三层：服务器证书。也就是跟我们服务器相关的这个证书了。

建立过程耗时，那么怎么优化呢？

1、升级HTTP2.0

> HTTP 2.0在2013年8月进行首次合作共事性测试。在开放互联网上HTTP 2.0将只用于https://网址，而 http://网址将继续使用HTTP/1，目的是在开放互联网上增加使用加密技术，以提供强有力的保护去遏制主动攻击

HTTP2主要有以下特性：

- 二进制分帧。数据使用二进制传输，相比于文本传输，更利于解析和优化。
- 多路复用。同域名下所有通信都在单个连接上完成，单个连接也可以承载任意数量的双向数据流。

- 头部优化。HTTP/2对消息头采用HPACK（专为http/2头部设计的压缩格式）进行压缩传输，能够节省消息头占用的网络的流量。

2、利用SessionID

这一点刚才已经说过了，为了在断开重连后，重复连接过程，所以使用SessionID记录会话id，然后就可以重新复用定位到哪个会话了。从而减去了重复发送证书、生成密钥过程。

3、TLS False Start

这是Google提出来的优化方案，具体做法是：

在TLS握手协商的第二个阶段，也就是客户端在验证证书，发送了pre—master secret之后，就直接把应用数据带上，比如请求网页数据。

然后服务器端收到pre—master secret后，生成对称密钥，然后直接用对称密钥解密这个应用数据，并响应消息给客户端。

其实就是把两个步骤混合为一个步骤了，客户端不需要等待服务器确认，再发送应用数据，而是直接在第二阶段就和pre—master secret一起发送给服务器端，减少了握手过程，从而减少了耗时。

4、OCSP Stapling

OCSP是一种验证检查证书吊销状态（合法性）的在线查询服务。

验证证书的过程中有一步是验证证书的合法性，我们可以让服务器先通过OCSP查询证书是否合法，然后把这个结果和证书一起发送给客户端，客户端就不需要单独验证证书的合法性了，从而提高了TLS握手效率。这个功能就叫做OCSP Stapling。

扩展：

如果不考虑建立过程，从整个Https传输过程考虑，又有哪些优化的点呢？

可以看看这篇文章介绍：*https://www.cnblogs.com/evan-blog/p/9898046.html*

讲一下HTTP和HTTPS的区别

经过上面大篇幅的讲解，对于两者的区别应该很明了了：

- HTTP是超文本传输协议，信息是明文传输，HTTPS则是在HTTP层下加了一层具有安全性的SSL/TLS加密传输协议，要用到CA证书。

- HTTP是没有身份认证的，客户端无法知道对方的真实身份。HTTPS加入了CA证书，可以确认对方信息。
- HTTP默认端口为80，HTTPS为443。
- HTTP因为明文传输，容易被攻击或者流量劫持。

怎么实现分块传输，断点续传？

分块传输

正常情况下，一次数据发完之后，服务器就会断开链接。

所以一般要在请求头中设置Connection字段的值为：keep-alive，表示维持连接不要断开，一直到某个数据包的Connection字段的值为close。

另外还有一种办法可以维持TCP连接，就是将请求数据进行分块传输。

分块传输指的是服务器发给客户端的数据可以分成多个部分传输。

使用方法：

- 消息头部设置Transfer-Encoding: chunked
- 每一块会表明长度
- 由一个标明长度为0的chunk标示结束

目的：

让客户端快速响应，减少等待时间。维持长连接。

但是、但是、这个分块传输只在HTTP1.1才有。HTTP2.0支持了多路复用，单个连接可以承载任意数量的双向数据流，也就是可以任意在一个连接在进行双向传输，不需要分块传输这个功能了。

断点续传

指的是客户端想从文件上次中断的地方开始下载或者上传,这样就算遇到网络问题导致下载或上传中断也没事了，保证好的用户体验。

使用方法：

- 客户端请求报文头部信息中加上Range字段，表示要从哪个字节开始下载，到哪个字节结束（Range: bytes=0-499）

- 服务器端响应报文头部信息中加上Content-Range，表示当前发送的数据的范围，以及文件总大小（Content-Range: bytes 0-499/22400）。
- ETag字段表示文件的唯一性。

实际使用流程：

- 第一次客户端请求下载，服务器端会返回文件内容，和Etag标示，状态码为200。
- 第二次客户端请求断点续传，会发送两个头部信息（Range:bytes=200-499，If-Range：Etag）。
- 然后服务器会判断Etag是否匹配，如果匹配则返回这一部分数据（Content-Range: bytes 200-499/22400），状态码为206，表示这是你请求的部分数据。否则会返回文件全部数据，状态码为200。

Http传输图片有哪些方式

其实这种问题问的是对于Content-Type的认识，一共三种方法：

multipart/form-data

表单类型传输文件请求。通过设置content-type为multipart/form-data，来发送二进制格式文件。支持多个文件上传，还可以带上文本参数。

这种是最常见的做法。

image/png，image/jpeg

这种方法就是直接将图片转为二进制流传输，服务器端也是直接读取流中的数据转成图片即可。

但是这种方法有个缺点就是一次只能传一张图片。

application/x-www-form-urlencoded，text/plain

还有个办法就是将图片转成Base64格式字符串，然后进行传输，和普通的文本参数一样，设置application/x-www-form-urlencoded或者text/plain等Content-Type即可。

参考

《网络是怎样连接的》

《Android开发艺术探索》

《Android进阶解密》

https://mp.weixin.qq.com/s/wy9V4wXUoEFZ6ekzuLJySQ

https://blog.csdn.net/weixin_43766753/article/details/108350589

https://wanandroid.com/wenda/show/12119

https://developer.android.google.cn/reference/android/os/Parcel?hl=en

https://juejin.cn/post/6854573218334769166#heading

https://www.zhihu.com/question/283510695

https://wetest.qq.com/lab/view/110.html

https://www.zhihu.com/question/271701044

https://www.cnblogs.com/wqhwe/p/5407468.html

http://www.ruanyifeng.com/blog/2017/06/tcp-protocol.html

https://network.51cto.com/art/201909/602938.htm

https://www.dazhuanlan.com/2019/11/21/5dd5aeeff1d0b/

https://zhuanlan.zhihu.com/p/26559480

http://gityuan.com/android/

https://www.jianshu.com/p/45cf56172d22

https://blog.csdn.net/itachi85/article/details/54783506

最后推荐一下我做的网站，方包博客: www.fang1688.cn ，包含详尽的知识体系、好用的工具，还有本公众号文章合