1. 谈谈你对 Android 设计模式的理解
2. 你所知道的设计模式有哪些？
3. 项目中常用的设计模式
4. 写出观察者模式
5. 的代码

https://blog.csdn.net/itachi85/article/details/50773358

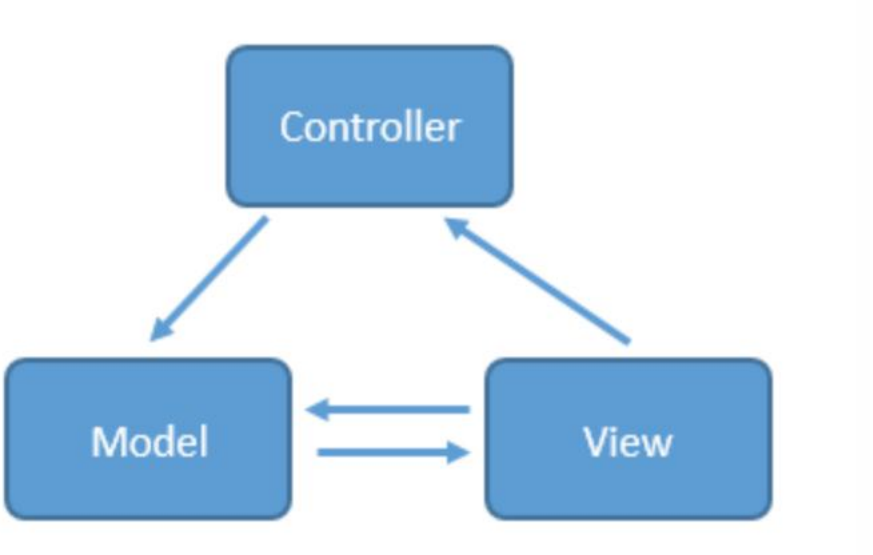
https://www.jianshu.com/p/1a9f571ad7c0

查看 23 中设计模式

6. MVC MVP MVVM 原理和区别

Android 鼓励弱耦合和组件的重用

MVC



主要组成部分：

1) **视图（View）**:用户界面。是应用程序中负责生成用户界面的部分。也是在整个 mvc 架构中用户唯一可以看到的一层，接收用户的输入，显示处理结果。(一般采用 xml 文件进行界面的描述，使用的时候可以非常方便的引入。)

2) **控制器（Controller）**:业务逻辑。是根据用户的输入，控制用户界面数据显示及更新 model 对象状态的部分，控制器更重要的一种导航功能,响应用户出发的相关事件,交给 m 层处理。(android 的控制层的重任通常落在了众多的 acitvity 的肩上,这句话也就暗含了不要在 acitivity 中写过多的代码，要通过 activity 交割 model 业务逻辑层处理，这样做的另外一个原因是 android 中的 acitivity 的响应时间是 5s，如果耗时的操作放在这里，程序就很容易被回收掉。)

3) **模型（Model）**:数据保存。是应用程序的主体部分，所有的业务逻辑都应该写在该层。(对数据库的操作、对网络等的操作都应该在 model 里面处理，当然对业务计算等操作也是必须放在的该层的。)

各部分之间的通信方式如下：

1. View 传送指令到 Controller。
2. Controller 完成业务逻辑后，要求 Model 改变状态。

3. Model 将新的数据发送到 View，用户得到反馈。

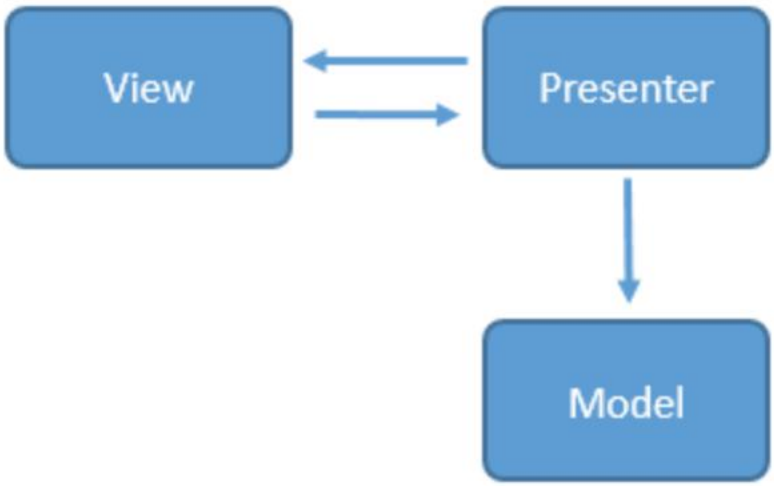
Tips：所有的通信都是单向的。MVC 还有一个重要的缺陷，view 层和 model 层是相互可知的，这意味着两层之间存在耦合，耦合对于一个大型程序来说是非常致命的，因为这表示开发，测试，维护都需要花大量的精力。

互动模式 接受用户指令时，MVC 可以分为两种方式。

1. 通过 View 接受指令，传递给 Controller。

2. 直接通过 Controller 接受指令

MVP



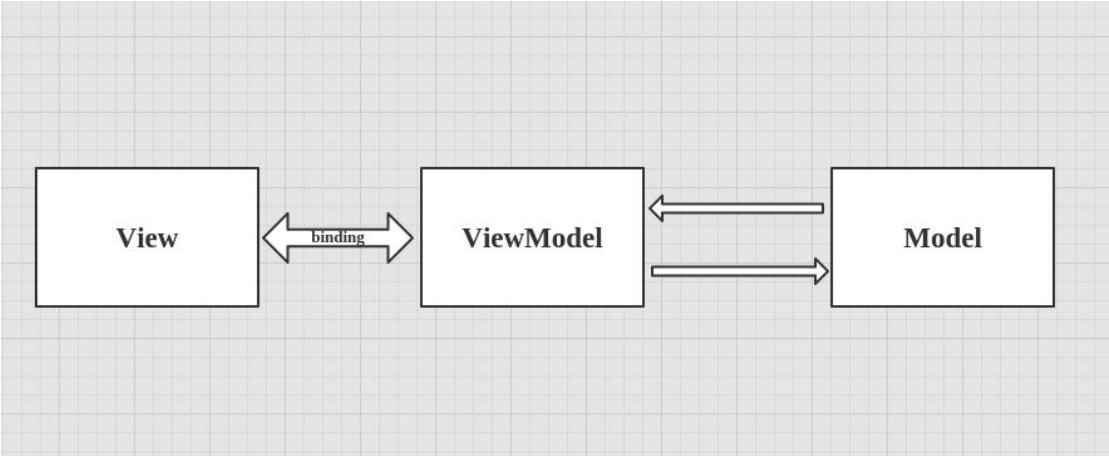
1. MVP 模式将 Controller 改名为 Presenter，同时改变了通信方向。

2. 各部分之间的通信，都是双向的

3. View 和 Model 不发生联系，都通过 Presenter 传递

4. View 非常薄，不部署任何业务逻辑，称为"被动视图"(Passive View)，即没有任何主动性，而 Presenter 非常厚，所有逻辑都部署在那里。

MVVM



MVVM 模式将 Presenter 改名为 ViewModel，基本上与 MVP 模式完全一致。可以说是 MVP 的升级版

唯一的区别是，它采用双向绑定(data-binding)：View 的变动，自动反映在 ViewModel，反之亦然。

7. 手写生产者/消费者模式

https://www.jianshu.com/p/a42b89287359

生产者/消费者

1) 生产者持续生产，直到缓冲区满，阻塞；缓冲区不满后，继续生产
2) 消费者持续消费，直到缓冲区空，阻塞；缓冲区不空后，继续消费
3) 生产者可以有多个，消费者也可以有多个

生产者和消费者，共用一个 BlockingQueue。为什么 BlockingQueue 能够实现生产者-消费者模型呢？对于 `put` 和 `take` 两个操作

Apple.java，生产和消费的对象。

```
public class Apple {

    private int id;

    public Apple(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Apple [id=" + id + "]";
    }
}
```

生产者：

```
public class Producer {
    BlockingQueue<Apple> queue;

    public Producer(BlockingQueue<Apple> queue) {
        this.queue = queue;
    }

    public boolean put(Apple apple) {
        return queue.offer(apple);
    }
}
```

消费者：

```
public class Consumer {
    BlockingQueue<Apple> queue;

    public Consumer(BlockingQueue<Apple> queue) {
        this.queue = queue;
    }

    public Apple take() throws InterruptedException {
        return queue.take();
    }
}
```

```
public class TestConsumer {

    public static void main(String[] args) {

        final BlockingQueue<Apple> queue = new LinkedBlockingDeque<Apple>(100);

        // 生产者
        new Thread(new Runnable() {

            int appleId = 0;
            Producer producer = new Producer(queue);

            @Override
            public void run() {
                try {
                    while (true) {
                        TimeUnit.SECONDS.sleep(1);
                        producer.put(new Apple(appleId++));
                        producer.put(new Apple(appleId++));
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

        // 消费者
        new Thread(new Runnable() {
            Consumer consumer = new Consumer(queue);

            @Override
            public void run() {
                try {
                    while (true) {
                        System.out.println(consumer.take().getId());
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

8. 适配器模式，装饰者模式，外观模

式的异同？

https://blog.csdn.net/AlbertFly/article/details/52276708

装饰者与适配者模式的区别

	适配器模式	装饰者模式
新职责/新功能	也可以增加新职责，但主要目的不在此	主要是给被修饰者增加新职责
原接口	用新接口调用原接口，原接口对新系统是不可见或者说不可用的	原封不动的使用原接口，系统对装饰的对象也通过原接口来完成使用
包裹的对象	是知道被适配者的详细情况	只知道接口是什么，至于其具体类型，只有在运行期间才知道

适配器模式将一个类的接口，转化成客户期望的另一个接口，适配器让原本接口不兼容的类可以合作无间。

装饰者模式：动态的将责任附加到对象上(因为利用组合而不是继承来实现，而组合是可以在运行时进行随机组合的)。若要扩展功能，装饰者提供了比继承更富有弹性的替代方案(同样地，通过组合可以很好的避免类暴涨，也规避了继承中的子类必须无条件继承父类所有属性的弊端)。

特点：

1. 装饰者和被装饰者拥有相同的超类型(可能是抽象类也可能是接口)

2. 可以用多个装饰类来包装一个对象，装饰类可以包装装饰类或被装饰对象

3. 因为装饰者和被装饰者拥有相同的抽象类型，因此在任何需要原始对象（被包装）的场合，都可以用装饰过的对象来替代它。

4. 装饰者可以在被装饰者的行为之前或之后，加上自己的附加行为，以达到特殊目的

5. 因为对象可以在任何的时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰对象

小结：装饰者模式——动态地将责任附加到对象上。想要扩展功能，装饰者提供了有别于继承的另外一种选择。是一个很好的符合了开闭原则的设计模式。

总结：适配器模式主要是为了接口的转换，而装饰者模式关注的是通过组合来动态的为被装饰者注入新的功能或行为(即所谓的责任)。

适配器将一个对象包装起来以改变其接口；装饰者将一个对象包装起来以增强新的行为和责任，而外观将一群对象包装起来以简化其接口

9. 用到的一些开源框架，介绍一个看

过源码的，内部实现过程。（？？？）

10. 谈谈对 RxJava 的理解

11. RxJava 的功能与原理实现

12. RxJava 的作用，与平时使用的异步操作来比的优缺点

https://www.jianshu.com/p/88aacbed8aa5

https://www.jianshu.com/p/f564104958b8

又叫响应式编程、响应式扩展 ReactiveX 是基于观察者模式设计的，核心对象只有 Observable 和 Observer。

Rx 使代码简化

1) **函数式风格：**对可观察数据流使用无副作用的输入输出函数，避免了程序里错综复杂的状态
2) **简化代码：**Rx 的操作符通通常可以将复杂的难题简化为很少的几行代码
3) **异步错误处理：**传统的 try/catch 没办法处理异步计算，Rx 提供了合适的错误处理机制
4) **轻松使用并发：**Rx 的 Observables 和 Schedulers 让开发者可以摆脱底层的线程同步和各种并发问题

框架源码有（待总结）

异步操作很关键的一点是程序的简洁性，因为在调度过程比较复杂的情况下，异步代码经常会既难写也难被读懂。 Android 创造的 AsyncTask 和 Handler ，其实都是为了让异步代码更加简洁。**RxJava 的优势也是简洁，但它的简洁的与众不同之处在于，随着程序逻辑变得越来越复杂，它依然能够保持简洁**（往往我们在使用异步任务的时候，使用 AsyncTask，但是后面业务变得复杂后，整个代码要改的很复杂，各种判断，头昏脑胀，但是有了 Rxjava 我们就能很简洁的实现代码逻辑了）。

13. 说说 EventBus 作用，实现方式，代替 EventBus 的方式

https://www.jianshu.com/p/334120fff804

EventBus 是什么

EventBus 是为 Android 优化的发布/订阅事件总线

1）简化组件之间的通信

2）分离事件发送者和接收者

3）对活动，片段和后台线程表现良好

4）避免复杂和容易出错的依赖关系和生命周期问题

5）使您的代码更简单

6）运行速度是快的

7）很小（约 50k 的 jar）

8）在实践中证明了具有 100,000,000+个安装的应用程序

9）具有传送线程，用户优先级等高级功能

RxJava 实现 RxBus

RXBus(RxBus 的核心功能是基于 Rxjava 的)

RxJava 写一个 RxBus，来实现 EventBus 的发布 / 订阅的事件总线功能。

或者按照 evenbus 原理来写：储存反射

14. 从 0 设计一款 App 整体架构，如何去做？

想要设计 App 的整体框架，首先要清楚我们做的是什么

一般我们与网络交互数据的方式有两种：主动请求(http)，长连接推送

结合网络交互数据的方式来说一下我们开发的 App 的类型和特点：

- **数据展示类型的 App**：特点是页面多，需要频繁调用后端接口进行数据交互，以 http 请求为主；推送模块，IM 类型 App 的 IM 核心功能以长连接为主，比较看重电量、流量消耗。
- **手机助手类 App**：主要着眼于系统 API 的调用，达到辅助管理系统的目的，网络调用的方式以 http 为主。
- **游戏**：一般分为游戏引擎和业务逻辑，业务脚本化编写，网络以长连接为主，http 为辅。

一般我们做的 App 都是类型 1，简要来说这类 app 的主要工作就是

1. 把服务端的数据拉下来给用户展示
2. 把用户在客户端修改的数据上传给服务端处理

所以这类 App 的网络调用相当频繁，而且需要考虑到网络差，没网络等情况下，App 的运行，成熟的商业应用的网络调用一般是如下流程：

UI 发起请求 - 检查缓存 - 调用网络模块 - 解析返回 JSON / 统一处理异常 - JSON 对象映射为 Java 对象 - 缓存 - UI 获取数据并展示

这之中可以看到很明显职责划分，即：数据获取；数据管理；数据展示

15. 谈谈对 java 状态机理解

https://glumes.com/post/android/understand-state-machine/

1. 有限状态机（FSM）

有限状态机（Finite State Machine）是表示有限个状态（State）以及在这些状态（State）之间的转移（Transition）和动作（Action）等行为的数据模型。

总的来说，有限状态机系统，是指在不同阶段呈现出不同的运行状态的系统，这些状态是有限的、不重叠的。

这样的系统在某一时刻一定会处于其所有状态中的一个状态，此时它接收一部分允许的输入，产生一部分可能的响应，并迁移到一部分可能的状态。

有限状态机的要素

1）State（状态）

状态（State），就是一个系统在其生命周期中某一个时刻的运行情况，此时，系统会执行一些操作，或者等待一些外部输入。并且，在当前形态下，可能会有不同的行为和属性。

2) Guard（条件）

状态机对外部消息进行响应时，除了需要判断当前的状态，还需要判断跟这个状态相关的一些条件是否成立。这种判断称为 Guard（条件）。Guard 通过允许或者禁止某些操作来影响状态机的行为。

3) Event（事件）

事件（Event），就是在一定的时间和空间上发生的对系统有意义的事情，事件通常会引起状态的变迁，促使状态机从一种状态切换到另一种状态。

4) Action（动作）

当一个事件（Event）被状态机系统分发的时候，状态机用 动作（Action）来进行响应，比如修改一下变量的值、进行输入输出、产生另外一个 Event 或者迁移到另外一个状态等。

5) Transition（迁移）

从一个状态切换到另一个状态被称为 Transition（迁移）。引起状态迁移的事件被称为触发事件（triggering event），或者被简称为触发（trigger）。

2. 分层状态机（HFSM）

在有限状态机中，虽说状态是有限的，但是当状态太多的时候却不是那么好维护的，这个时候就需要将一些具有公共属性的状态分类，抽离出来，将同类型的状态作为一个状态机，然后再做一个大的

状态机，来维护这些子状态。

例如，在有限状态机中，我们的状态图是这样的：

但在分层状态机中，我们的状态是这样的：

这样一来，我们就不需要考量所有的状态之间的关系，定义所有状态之间的跳转链接。

只需要使用层次化的状态机，将所有的行为分类，把几个小的状态归并到一个大的状态里面，然后再定义高层状态和高层状态中内部小状态的跳转链接。

分层状态机从某种程度上就是限制了状态机的跳转，而且高层状态内部的状态是不需要关心外部状态的跳转的，这也做到了无关状态间的隔离，在每个状态内部只需要关心自己的小状态的跳转就可以了，这样就大大的降低了状态机的复杂度。

16. Fragment 如果在 Adapter 中使用应该如何解耦？

startActivityForResult 是哪个类的方法,在什么情况下使用,如果在 Adapter 中使用应该如何解耦？（是这个？）

startActivityForResult 是 Activity 类里的方法，在原 Activity 里通过 Intent 跳转到其他类再跳回到原 Activity 里时候,回传数据所用

在 Adapter 以及其他非 Activity 类使用的时候,可以将由原

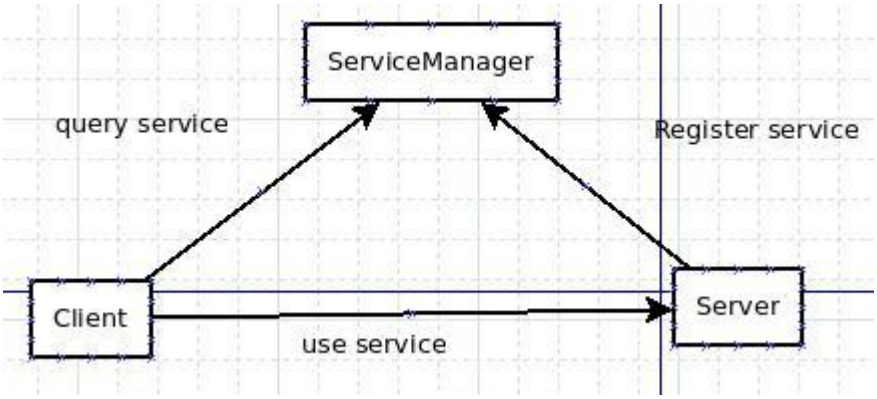
Activity 类传入的 Context 强转为 Activity 类,再在原 Activity 里重写 onActivityResult 方法接受到返回值。

17. Binder 机制及底层实现

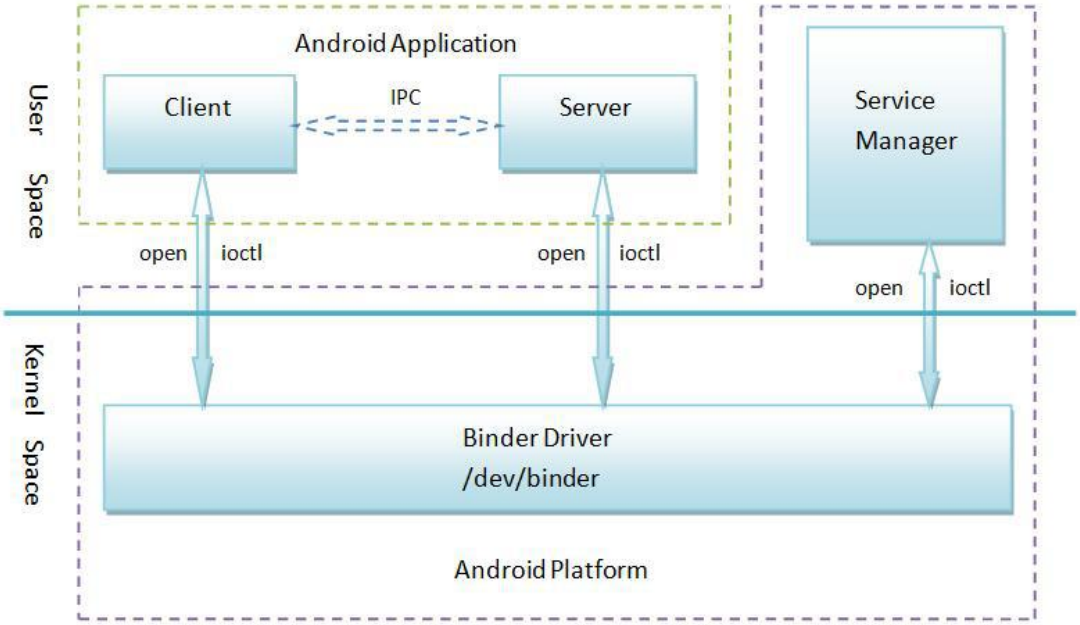
https://blog.csdn.net/weijinqian0/article/details/52233529

Android 采用 Binder 机制是有道理的。既然 Binder 机制这么多优点，那么我们接下来看看它是怎样通过 C/S 模型来实现的。

Binder 在 C/S 中的流程如下：



Binder 通信机制流程(整体框架)



上图即是 Binder 的通信模型。我们可以发现：

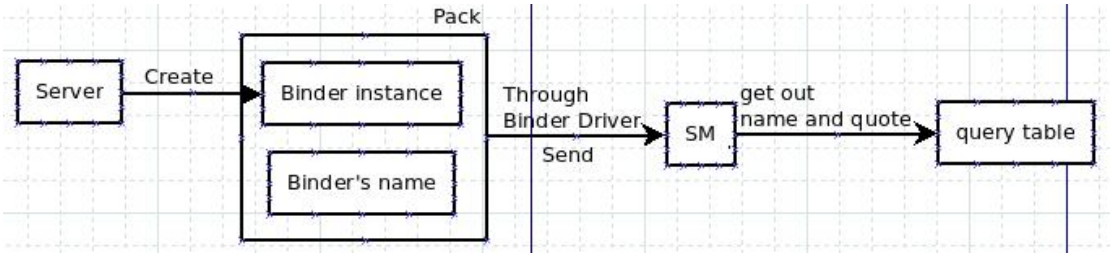
1) Client 和 Server 是存在于用户空间

2) Client 与 Server 通信的实现，是由 Binder 驱动在内核空间实现

3) SM 作为守护进程，处理客户端请求，管理所有服务项。

为了方便理解，我们可以把 SM 理解成 DNS 服务器；那么 Binder Driver 就相当于路由的功能。

步骤一：Server 向 SM 注册服务



1) 首先，XXXServer(XXX 代表某个)在自己的进程中向 Binder 驱动申请创建一个 XXXService 的 Binder 的实体(可以理解成具有真

实空间的 Object）

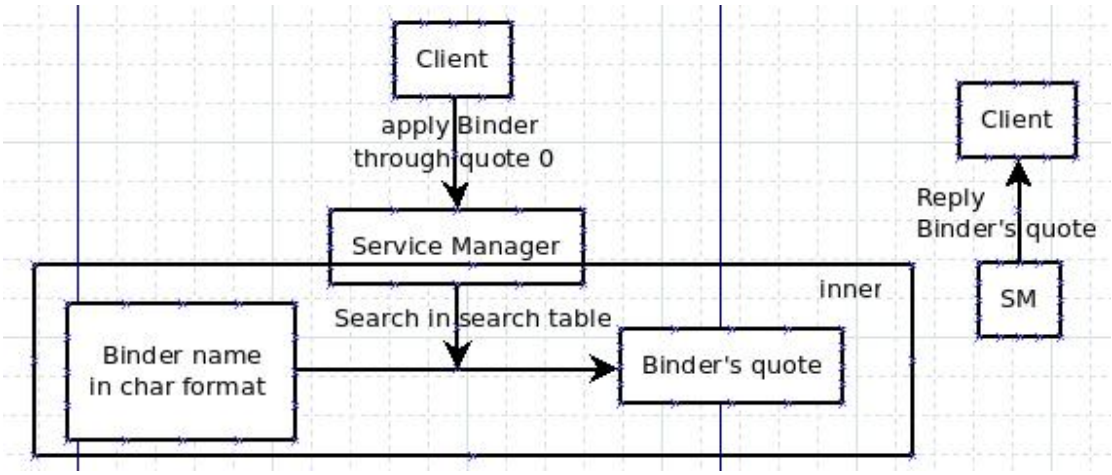
2） Binder 驱动为这个 XXXService 创建位于内核中的 Binder 实体节点以及 Binder 的引用（没有真实空间，可以理解成实体的 一个链接，操作引用就会操作对应链接上的实体），注意，是将名字和新建的引用打包传递给 SM（实体没有传给 SM），通知 SM 注册一个名叫 XXX 的 Service。

3） SM 收到数据包后，从中取出 XXXService 名字和引用，填入一张查找表中。

4） 此时，如果有 Client 向 SM 发送申请服务 XXXService 的请求，那么 SM 就可以在查找表中找到该 Service 的 Binder 引用，并把 Binder 引用(XXXBpBinder)返回给 Client。

注：引用和实体。这里，对于一个用于通信的实体，可以有多个该实体的引用。如果一个进程持有某个实体，其他进程也想操作该实体，最高效的做法是去获得该实体的引用，再去操作这个引用。
有些资料把实体称为本地对象，引用成为远程对象。可以这么理解：引用是从本地进程发送给其他进程来操作实体之用，所以有本地和远程对象之名。

步骤二：Client 从 SM 获得 Service 的远程接口



Server 向 SM 注册了 Binder 实体及其名字后，Client 就可以通过 Service 的名字在 SM 的查找表中获得该 Binder 的引用了（BpBinder）。

1) Client 也利用保留的 handle 值为 0 的引用向 SM 请求访问某个 Service：我申请访问 XXXService 的引用。

2) SM 就会从请求数据包中获得 XXXService 的名字，在查找表中找到该名字对应的条目，取出 Binder 的引用打包回复给 client。

之后，Client 就可以利用 XXXService 的引用使用 XXXService 的服务了。如果有更多的 Client 请求该 Service，系统中就会有更多的 Client 获得这个引用。

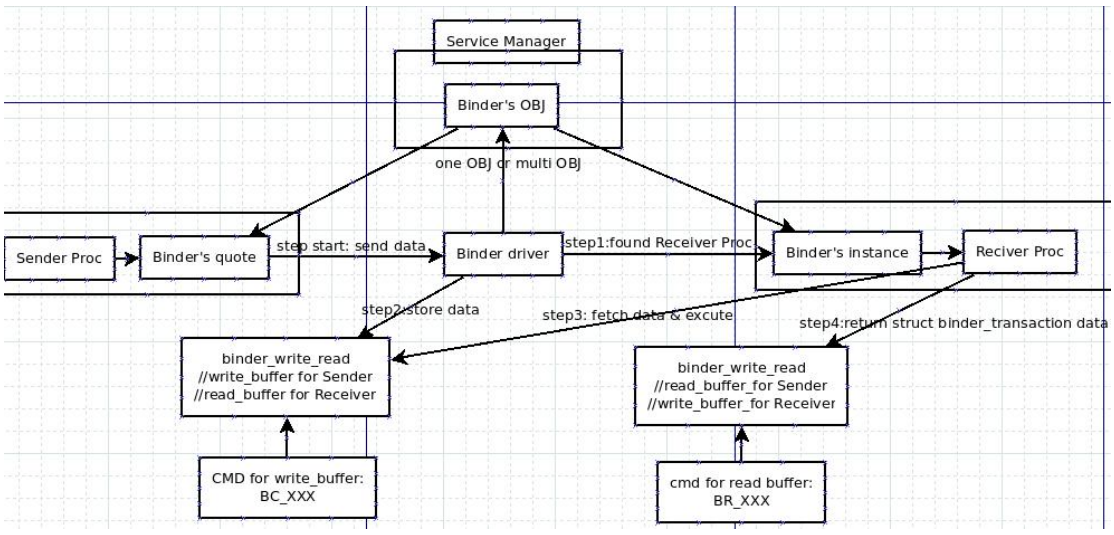
建立 C/S 通路后

首先要理清一个概念：client 拥有自己 Binder 的实体，以及 Server 的 Binder 的引用；Server 拥有自己 Binder 的实体，以及 Client 的 Binder 的引用。我们也可以从接收方和发送方的方式来理解：

1） 从 client 向 Server 发数据:Client 为发送方,拥有 Binder 的实体；Server 为接收方，拥有 Binder 的引用

2） 从 server 向 client 发数据:Server 为发送方,拥有 Binder 的实体；client 为接收方，拥有 Binder 的引用。

也就是说，我们在建立了 C/S 通路后，无需考虑谁是 Client 谁是 Server，只要理清谁是发送方谁是接收方，就能知道 Binder 的实体和引用在哪边。



建立 CS 通路后的流程：（当接收方获得 Binder 的实体，发送方获得 Binder 的引用后）

1）发送方会通过 Binder 实体请求发送操作。

2）Binder 驱动会处理这个操作请求,把发送方的数据放入写缓存

(binder_write_read.write_buffer)（对于接收方为读缓冲区），并把 read_size(接收方读数据)置为数据大小（对于具体的实现后面会介绍）；

3) 接收方之前一直在阻塞状态中，当写缓存中有数据，则会读取数据，执行命令操作

4) 接收方执行完后,会把返回结果同样用binder_transaction_data结构体封装，写入写缓冲区（对于发送方，为读缓冲区）

备注：

C/S （Client/Server）结构

即大家熟知的客户机和服务器结构，它是软件系统体系结构，通过它可以充分利用两端硬件环境的优势，将任务合理分配到 Client 端和 Server 端来实现，降低了系统的通讯开销。目前大多数应用软件系统都是 Client/Server 形式的两层结构，即客户端服务器端架构，其客户端包含一个或多个在用户的电脑上运行的程序，而服务器端有两种，一种是数据库服务器端，客户端通过数据库连接访问服务器端的数据；另一种是 Socket 服务器端，服务器端的程序通过 Socket 与客户端的程序通信。

- 优点：
  1、C/S 架构的界面和操作可以很丰富。
  2、安全性能可以很容易保证，实现多层认证也不难。
  3、由于只有一层交互，因此响应速度较快。
- 缺点：
  1、适用面窄，通常用于局域网中。
  2、用户群固定。由于程序需要安装才可使用，因此不适合面向一些不可知的用户。
  3、维护成本高，发生一次升级，则所有客户端的程序都需要改变。

18. 对于应用更新这块是如何做的？(解答：灰度，强制更新，分区域更新)？

灰度：（能够平滑过渡的一种发布方式）

(1)找单一渠道投放特别版本。

(2)做升级平台的改造，允许针对部分用户推送升级通知甚至版本强制升级。

(3)开放单独的下载入口。

还有，灰度版最好有收回的能力，一般就是强制升级下一个正式版

增量更新：bsdiff：二进制差分工具 bspatch 是相应的补丁合成工具，根据两个不同版本的二进制文件，生成补丁文件.patch 文件。通过 bspatch 使旧的 apk 文件与不定文件合成新的 apk。 不足：要区分版本，内置及版本相同破解版 apk 无法增量更新，最好进行 shalsum 校验，保证基础包的一致性。

19. 实现一个 Json 解析器(可以通过正

则提高速度）

https://zhuanlan.zhihu.com/p/28049617

可能要学会自己写

暂时不写

JSON 数据是递归的。并且，JSON 格式里是没有环的，实际上，JSON 格式表达了一颗树，一颗多叉树！

20. 统计启动时长,标准

https://blog.csdn.net/beyond_liyy/article/details/52273740

通常来说，在安卓中应用的启动方式分为以下几种：

1. **冷启动**：当启动应用时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用，这个启动方式就是冷启动。冷启动因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化 Application 类，再创建和初始化 MainActivity 类，最后显示在界面上。

2. **热启动**：当启动应用时，后台已有该应用的进程（例：按 back 键、home 键，应用虽然会退出，但是该应用的进程是依

然会保留在后台，可进入任务列表查看），所以在已有进程的情况下，这种启动会从已有的进程中来启动应用，这个方式叫热启动。热启动因为会从已有的进程中来启动，所以热启动就不会走 Application 这步了，而是直接走 MainActivity，所以热启动的过程不必创建和初始化 Application，因为一个应用从新进程的创建到进程的销毁，Application 只会初始化一次。

3. **首次启动：**首次启动严格来说也是冷启动，之所以把首次启动单独列出来，一般来说，首次启动时间会比非首次启动要久，首次启动会做一些系统初始化工作，如缓存目录的生产，数据库的建立，SharedPreference 的初始化，如果存在多 dex 和插件的情况下，首次启动会有一些特殊需要处理的逻辑，而且对启动速度有很大的影响，所以首次启动的速度非常重要，毕竟影响用户对 App 的第一映像。

一、本地启动时间的统计方式

如果是本地调试的话，统计启动时间还是很简单的，通过命令行方式即可：

```
adb shell am start -w packagename/activity
```

输出的结果类似于：

```
$ adb shell am start -W com.speed.test/com.speed.test.HomeActivity
```

```
Starting: Intent { act=android.intent.action.MAIN
cat=[android.intent.category.LAUNCHER] cmp=com.speed.test/.HomeActivity }
Status: ok
Activity: com.speed.test/.HomeActivity
ThisTime: 496
TotalTime: 496
WaitTime: 503
Complete
```

WaitTime 返回从 startActivity 到应用第一帧完全显示这段时间. 就是总的耗时，包括前一个应用 Activity pause 的时间和新应用启动的时间；

ThisTime 表示一连串启动 Activity 的最后一个 Activity 的启动耗时；

TotalTime 表示新应用启动的耗时，包括新进程的启动和 Activity 的启动，但不包括前一个应用 Activity pause 的耗时。

开发者一般只要关心 TotalTime 即可，这个时间才是自己应用真正启动的耗时。

二、线上启动时间的统计方式

当 App 发到线上之后，想要统计 App 在用户手机上的启动速度，就不能通过命令行的方式进行统计了，基本上都是通过打 Log 的方式将启动时间发送上来。那么在什么位置加启动时间统计的 Log 就尤为重要，Log 添加的位置直接决定启动时间统计的是否准确，同样也会影响启动速度优化效果的判断。要想找到合适准确的位置记录启动时间的 Log，就需要了解应用的启动流程，和各个生命周

期函数的调用顺序。下面来分析下到底在什么位置打 Log 记录启动时间比较合适。

应用的主要启动流程

关于 App 启动流程的文章很多，文章底部有一些启动流程相关的参考文章，这里只列出大致流程如下：

1. 通过 Launcher 启动应用时，点击应用图标后，Launcher 调用 *startActivity* 启动应用。
2. Launcher Activity 最终调用 *Instrumentation* 的 *execStartActivity* 来启动应用。
3. *Instrumentation* 调用 *ActivityManagerProxy* (*ActivityManagerService* 在应用进程的一个代理对象) 对象的 *startActivity* 方法启动 *Activity*。
4. 到目前为止所有过程都在 Launcher 进程里面执行，接下来 *ActivityManagerProxy* 对象跨进程调用 *ActivityManagerService*（运行在 *system_server* 进程）的 *startActivity* 方法启动应用。
5. *ActivityManagerService* 的 *startActivity* 方法经过一系列调用，最后调用 *zygoteSendArgsAndGetResult* 通过 *socket* 发送给 *zygote* 进程，*zygote* 进程会孵化出新的应用进程。

6. zygote 进程孵化出新的应用进程后，会执行 ActivityThread 类的 main 方法。在该方法里会先准备好 Looper 和消息队列，然后调用 attach 方法将应用进程绑定到 ActivityManagerService，然后进入 loop 循环，不断地读取消息队列里的消息，并分发消息。
7. ActivityManagerService 保存应用进程的一个代理对象，然后 ActivityManagerService 通过代理对象通知应用进程创建入口 Activity 的实例，并执行它的生命周期函数。

总结过程就是：用户在 Launcher 程序里点击应用图标时，会通知 ActivityManagerService 启动应用的入口 Activity， ActivityManagerService 发现这个应用还未启动，则会通知 Zygote 进程孵化出应用进程，然后在这个应用进程里执行 ActivityThread 的 main 方法。应用进程接下来通知 ActivityManagerService 应用进程已启动，ActivityManagerService 保存应用进程的一个代理对象，这样 ActivityManagerService 可以通过这个代理对象控制应用进程，然后 ActivityManagerService 通知应用进程创建入口 Activity 的实例，并执行它的生命周期函数。

生命周期函数执行流程

上面的启动流程是 Android 提供的机制，作为开发者我们需要清楚或者至少了解其中的过程和原理，但我们并不能在这过程中做什么文章，我们能做的恰恰是从上述过程中最后一步开始，即 ActivityManagerService 通过代理对象通知

应用进程创建入口 Activity 的实例，并执行它的生命周期函数开始，我们的启动时间统计以及启动速度优化也是从这里开始。下面是 Main Activity 的启动流程：

```
-> Application 构造函数
-> Application.attachBaseContext()
-> Application.onCreate()
-> Activity 构造函数
-> Activity.setTheme()
-> Activity.onCreate()
-> Activity.onStart
-> Activity.onResume
-> Activity.onAttachedToWindow
-> Activity.onWindowFocusChanged
```

如果打 Log 记录 App 的启动时间，那么至少要记录两个点，一个起始时间点，一个结束时间点。

(1) 起始时间点

起始时间点比较容易记录：

1）如果记录冷启动启动时间一般可以在 Application.attachBaseContext() 开始的位置记录起始时间点，因为在这之前 Context 还没有初始化，一般也干不了什么事情，当然这个是要视具体情况来定，其实只要保证在 App 的具体业

务逻辑开始执行之前记录起始时间点即可。

2) 如果记录热启动启动时间点可以在 Activity.onRestart() 中记录起始时间点。

(2) 结束时间点

1) **不应该在 onResume 方法执行完成后记录：** Activity 的 onResume 方法执行完成之后，Activity 就对用户可见了，实际上并不是，一个 Activity 走完 onCreate onStart onResume 这几个生命周期之后，只是完成了应用自身的一些配置,比如 Activity 主题设置 window 属性的设置 View 树的建立，但是其实后面还需要各个 View 执行 measure layout draw 等。所以在 OnResume 中记录结束时间点的 Log 并不准确

2) **我们可以在 Activity.onWindowFocusChanged 记录应用启动的结束时间点**，不过需要注意的是该函数，在 Activity 焦点发生变化时就会触发，所以要做好判断，去掉不需要的情况。