Android开发中为人所忽视的异步编程模式

Charles Zhang | 张铁蕾

About Me

- Charles Zhang | 张铁蕾
- CTO@微爱
- Programming Experience 10 Years
- Blog: zhangtielei.com

“

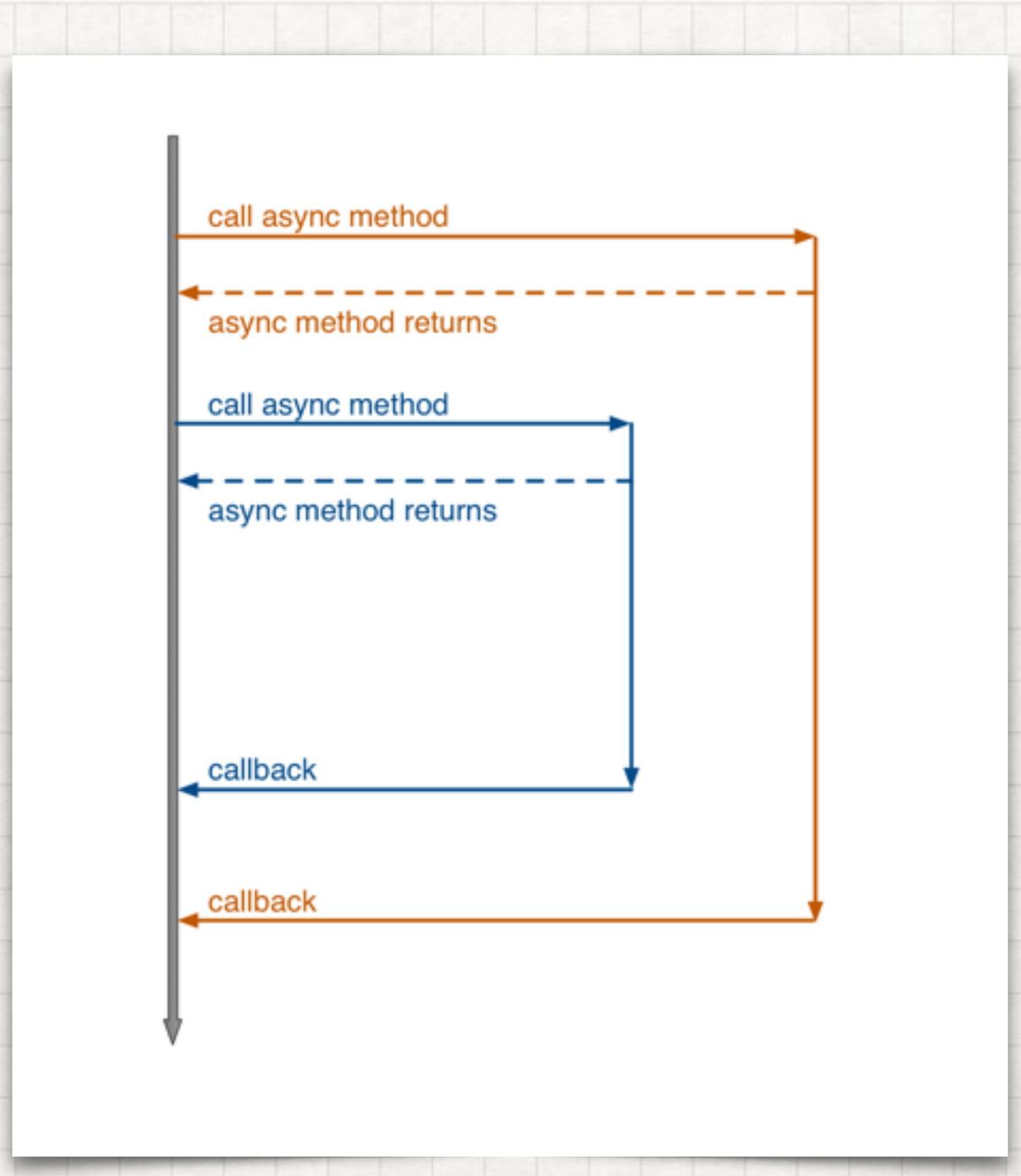
A common principle throughout computer science is that we can build complicated systems from minimal components.

— *Ian Goodfellow and Yoshua Bengio and Aaron Courville*

”

什么是异步编程?

- 异步任务
  - 偏离主程序流程
  - 不用等待任务结束
- 异步事件
  - 可以在任何时间发生
  - 例如 BroadcastReceiver



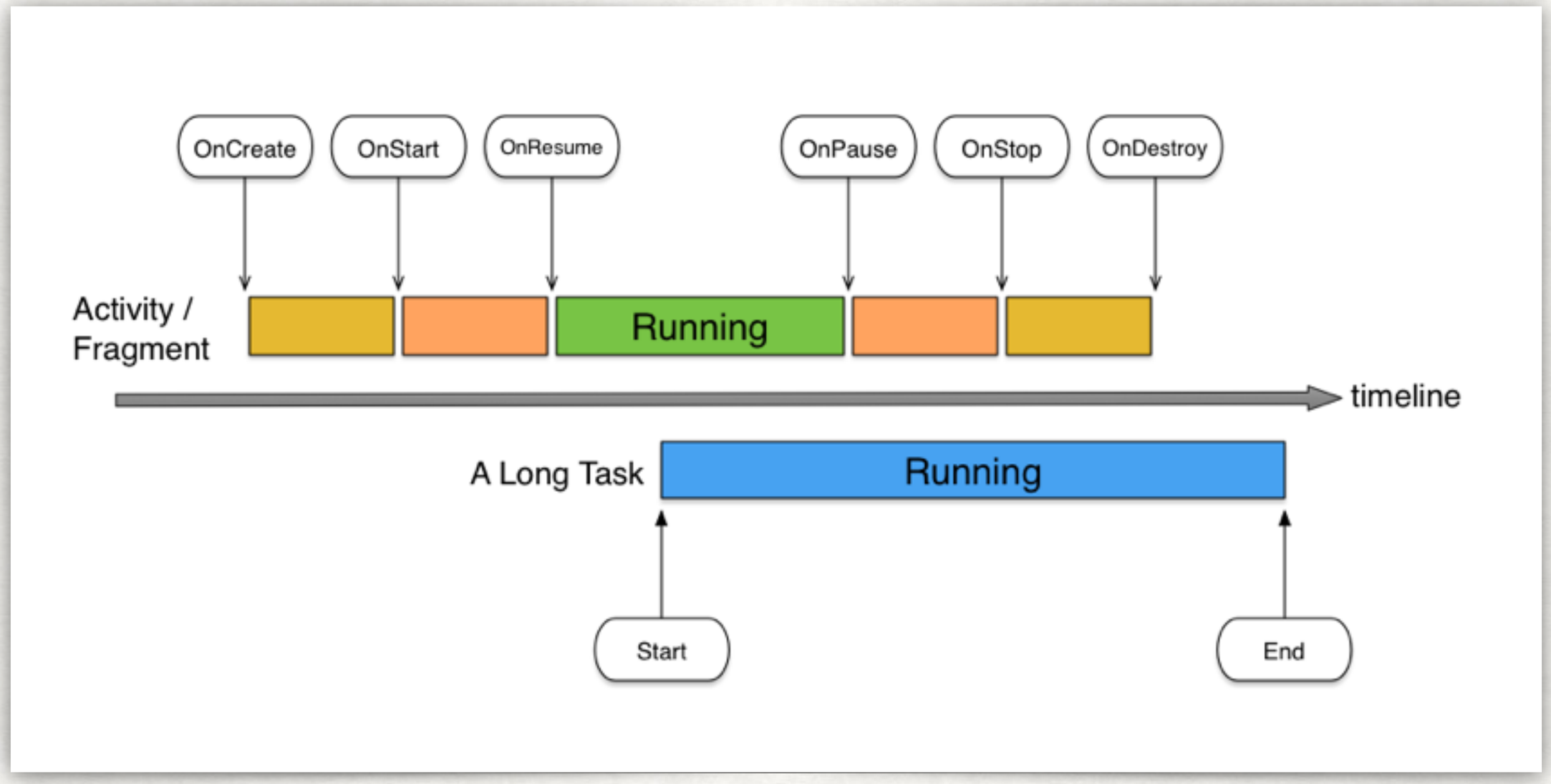
Android中的异步编程

- 「前端」编程 (UI)
  - Layout
  - 事件处理
  - 绘制渲染
- 「后端」编程 (逻辑)
  - 数据存储和缓存
  - 队列
  - 网络编程
  - 生命周期管理

	Android Programming	
	Frontend (UI)	Backend (Logic)
Async Programming	★★★	★★ ★★★

异步编程与「前后端」编程的相关性

举例：Fragment 提前销毁



- 长任务的例子
  - 网络请求，下载，上传，等等

举例：长任务结束时的风险

- 对于Fragment
  - getActivity() 返回 null.
  - getView() 返回 null.
    - getResources().getString(id) 抛异常 IllegalStateException.
- 对于Activity / Fragment
  - 可能造成内存无法释放（持续到异步长任务结束）

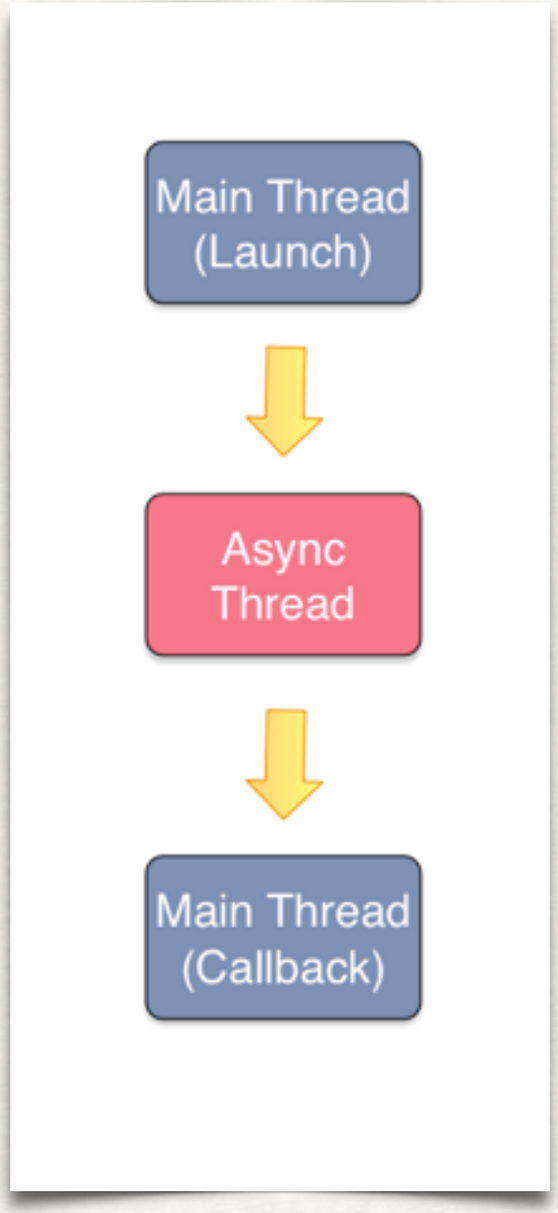
举例：可能的解决思路

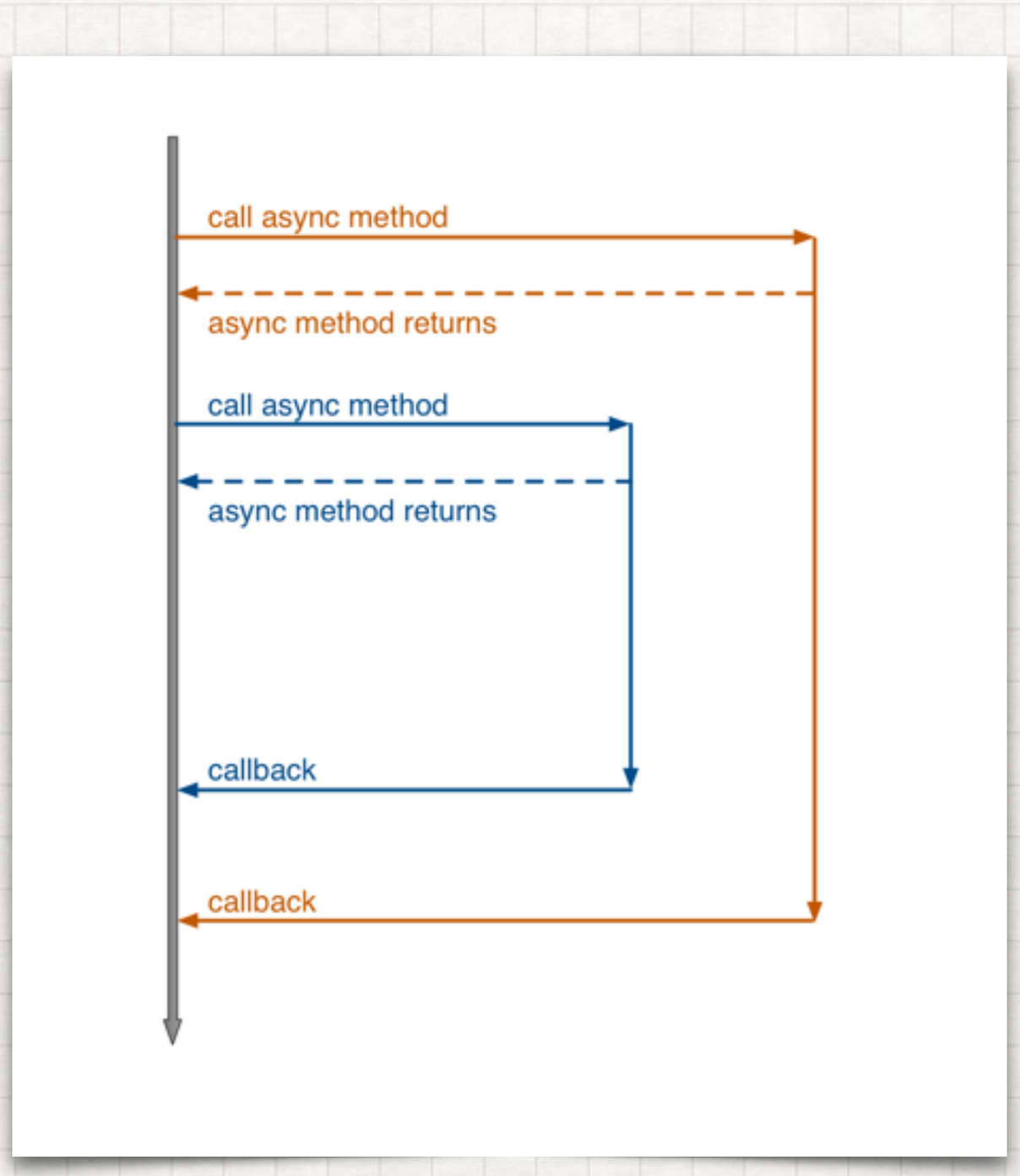
- Activity / Fragment 销毁时取消任务
- 忽略任务执行结果
- Bind / Unbind

在Android上启动异步任务的方式

- 网络请求
  - 例如 Volley, Retrofit
- 线程池调度
  - 例如 ExecutorService, ScheduledExecutorService
- Looper调度
  - 例如 Handler.post/sendMessage, View.postDelayed, Handler.postDelayed/postAtTime
- 系统相关的异步行为
  - 例如 startActivity, launching a fragment

异步任务的线程模型


Main Thread
(Launch)
Async
Thread
Main Thread
(Callback)


call async method
async method returns
call async method
async method returns
callback
callback

回调到主线程！

异步编程模式总结

I. 多个异步任务协作

II. 异步任务与队列

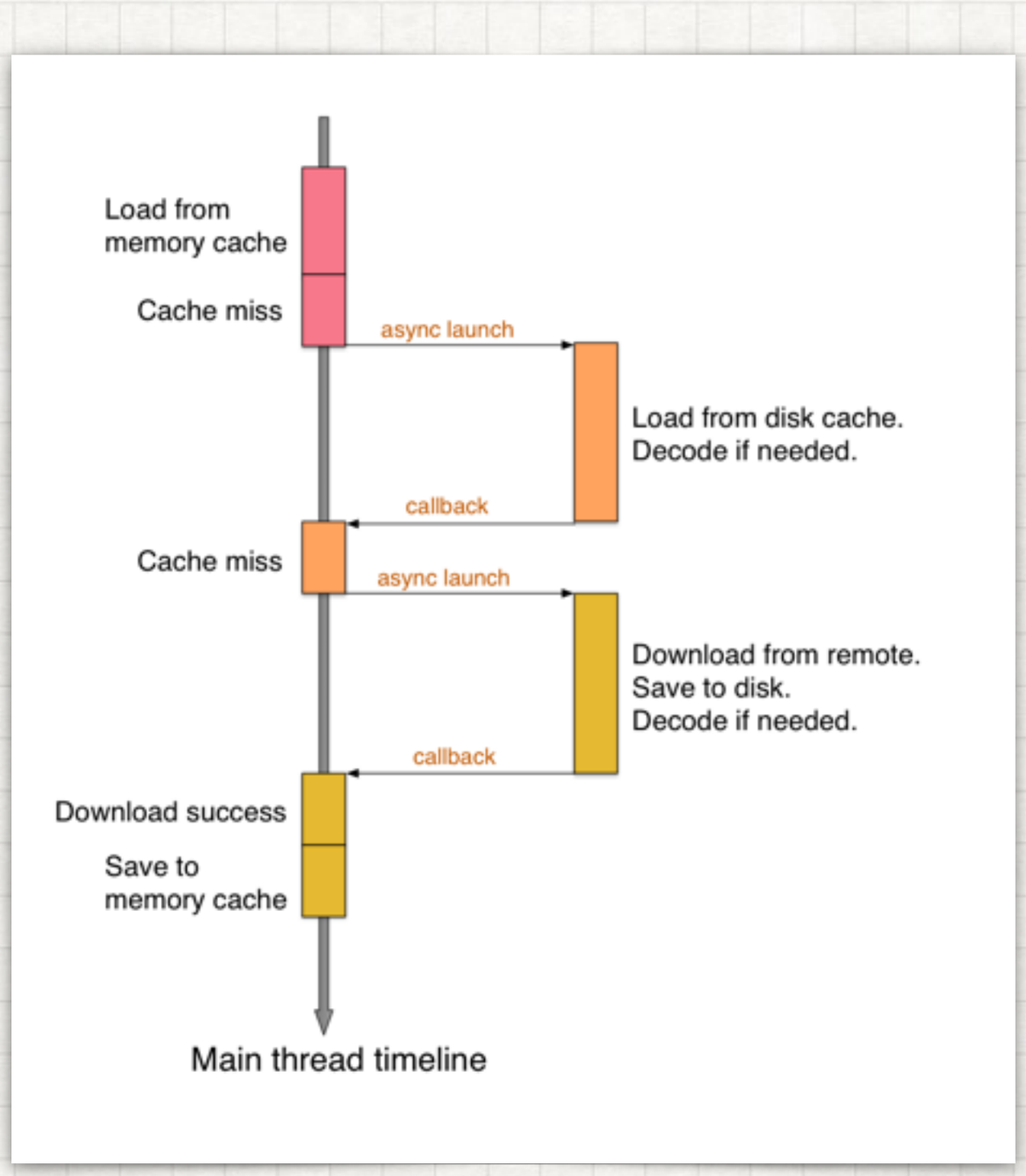
III. 异步任务的取消

多个异步任务的协作关系

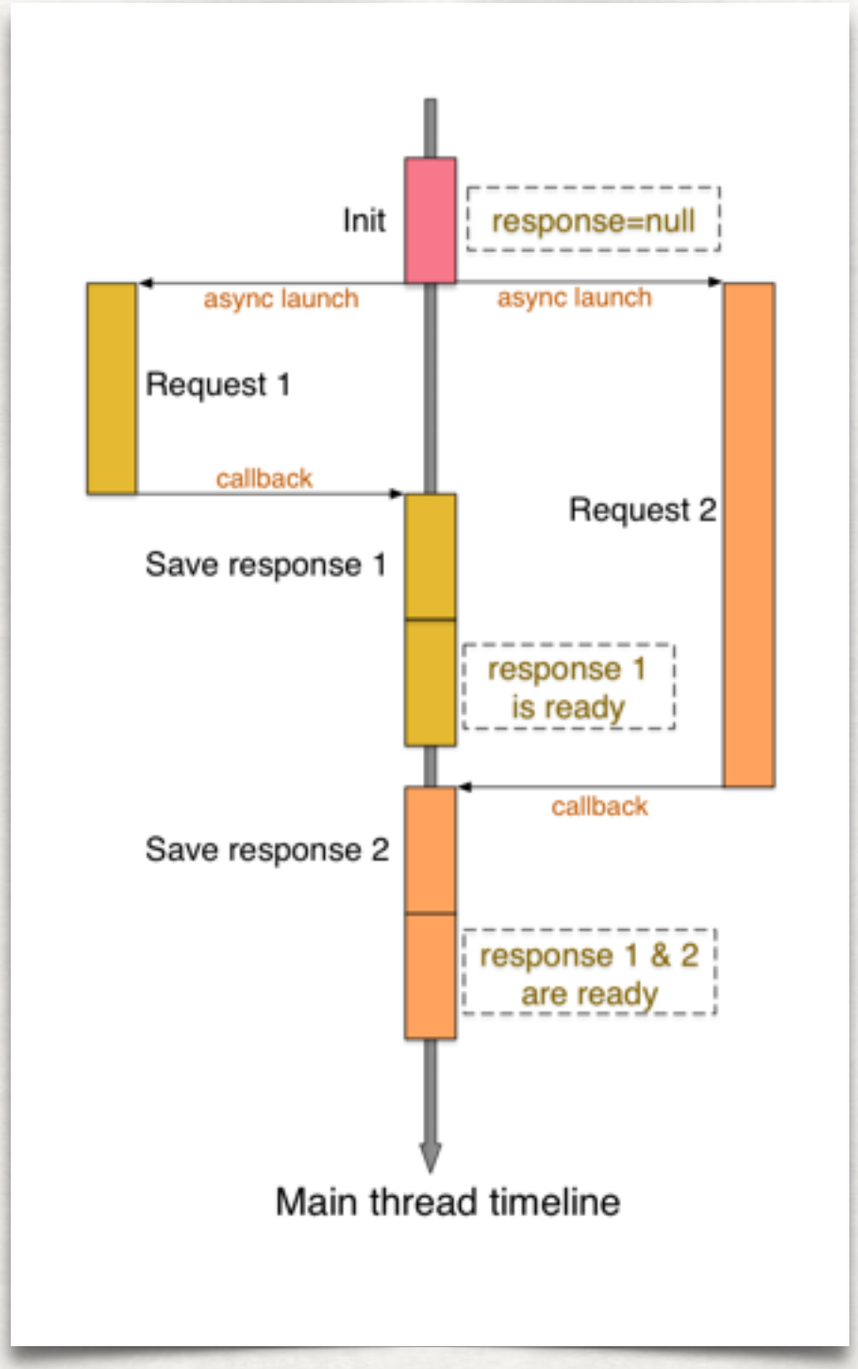
- 先后接续执行
  - 举例 多级缓存
- 并发执行，结果合并
  - 举例 并发网络请求
- 并发执行，一方优先
  - 举例 简单的页面缓存

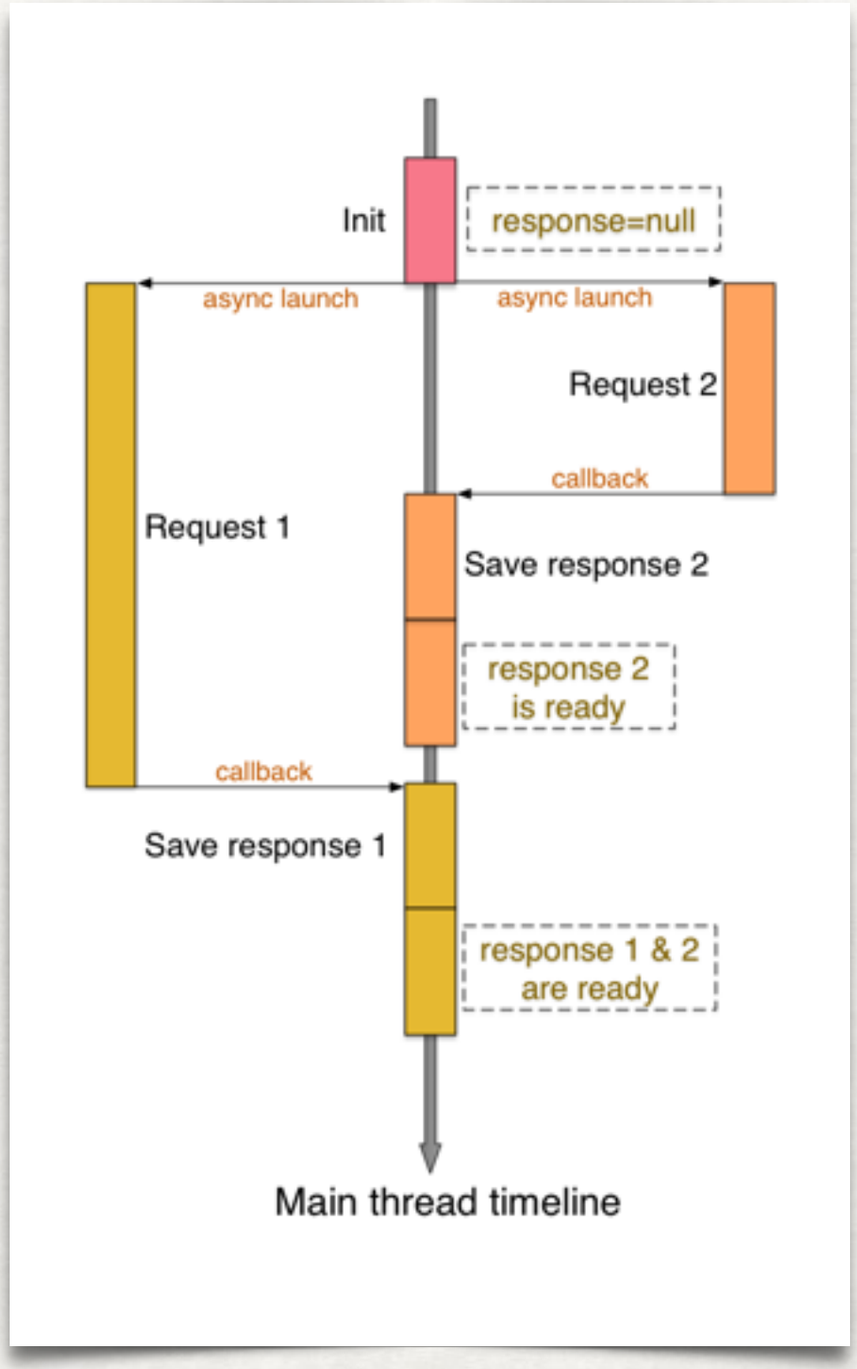
多级缓存 / 接续执行

- Memory Cache (同步)
- Disk Cache (异步)
- 从网络下载 (异步)



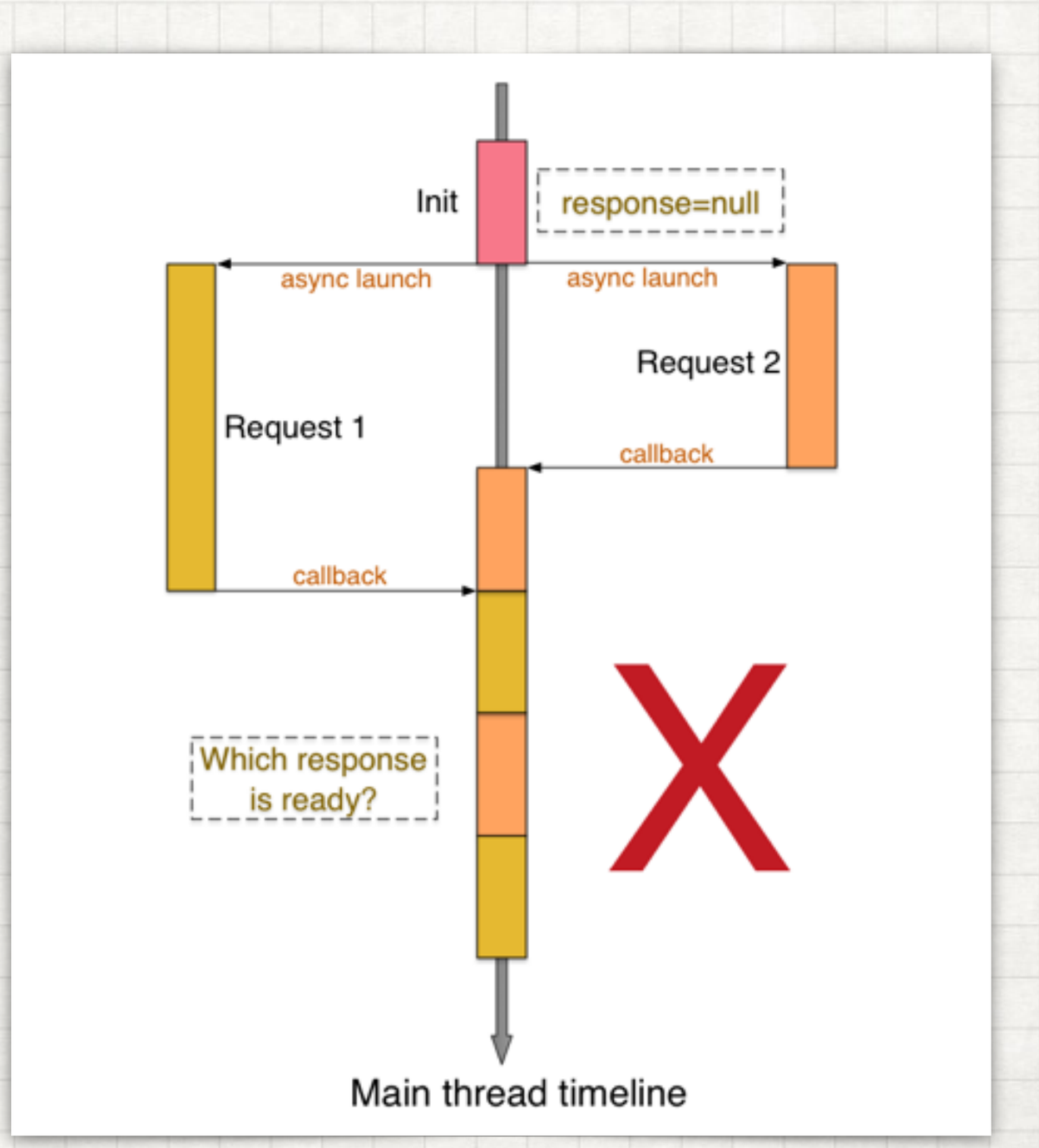
并发网络请求 / 合并


Init
response=null
async launch
async launch
Request 1
callback
Request 2
Save response 1
response 1
is ready
callback
Save response 2
response 1 & 2
are ready
Main thread timeline

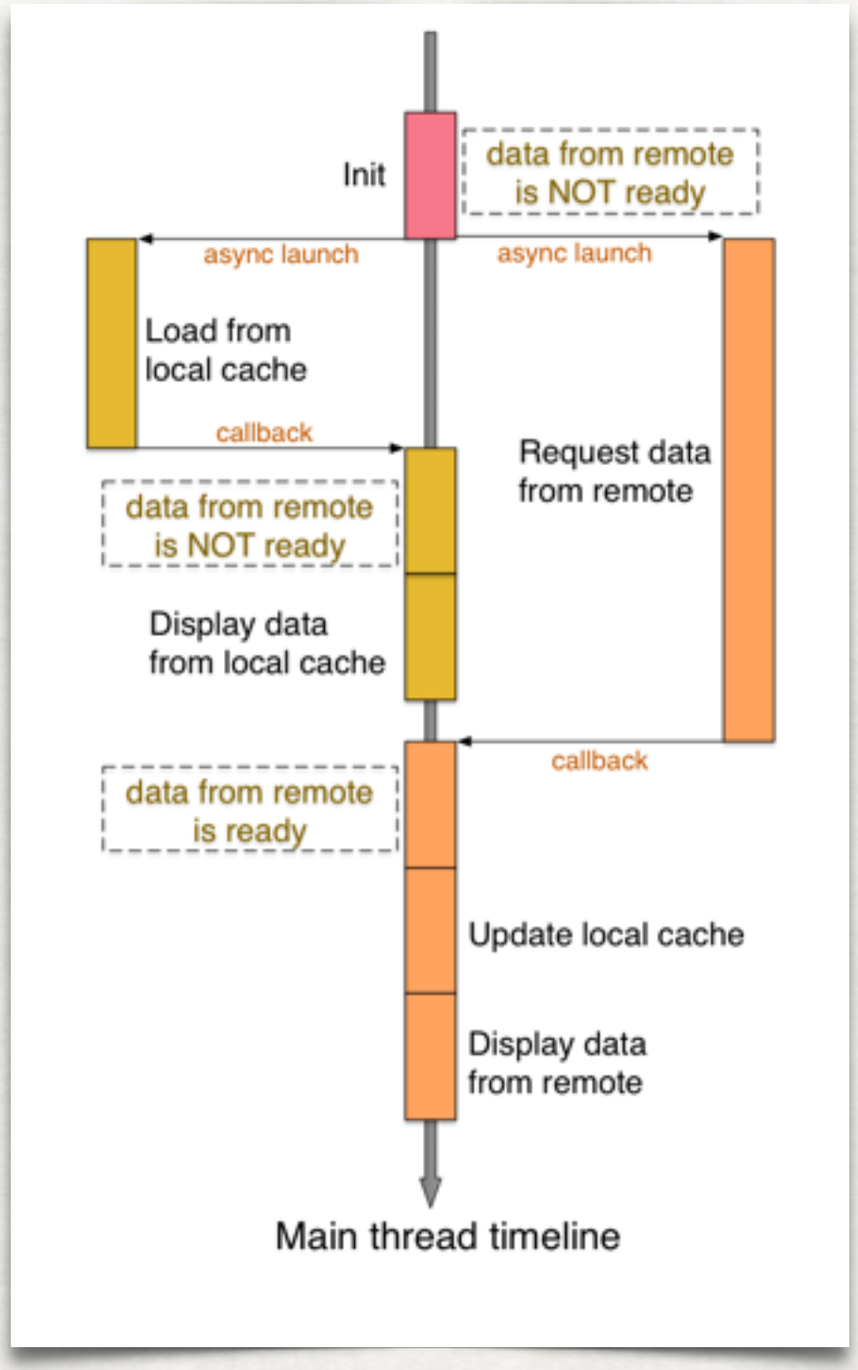

Init
response=null
async launch
async launch
Request 2
callback
Request 1
Save response 2
response 2
is ready
callback
Save response 1
response 1 & 2
are ready
Main thread timeline

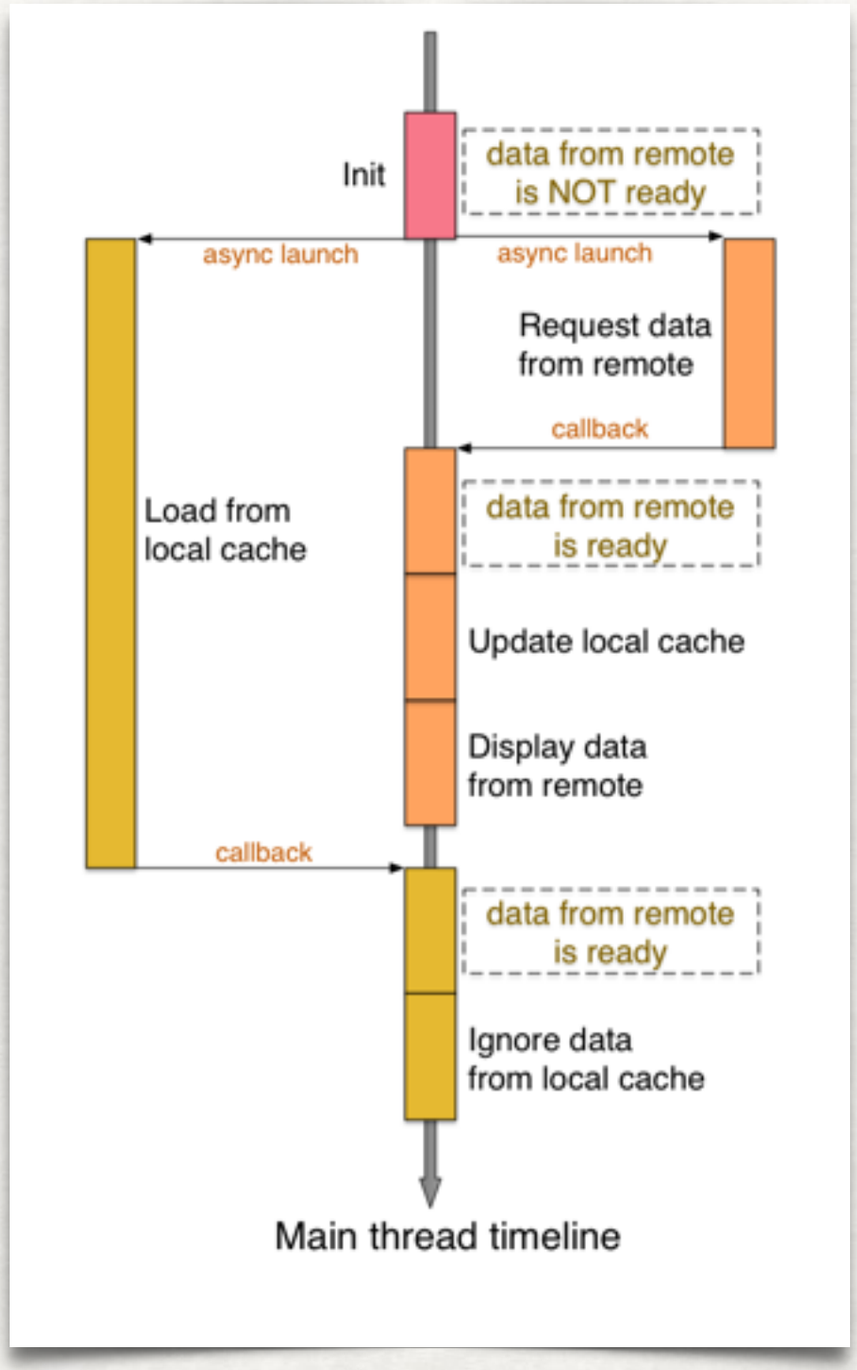
并发网络请求 / 合并

- 不可能在同一个主线程上交叉执行代码!



页面缓存 / 一方优先

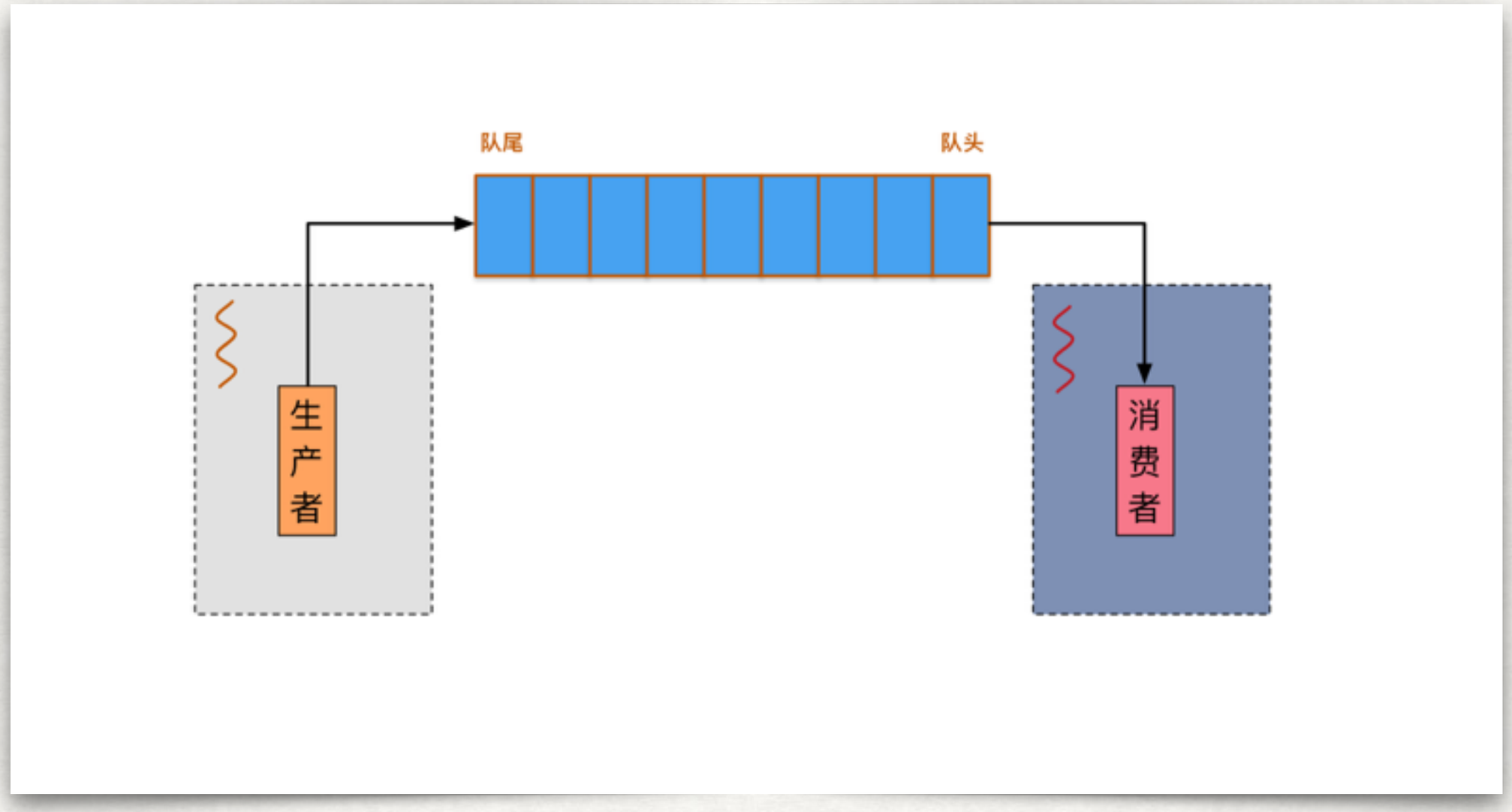

Init
data from remote
is NOT ready
async launch
async launch
Load from
local cache
callback
Request data
from remote
data from remote
is NOT ready
Display data
from local cache
callback
data from remote
is ready
Update local cache
Display data
from remote
Main thread timeline


Init
data from remote
is NOT ready
async launch
async launch
Request data
from remote
callback
Load from
local cache
data from remote
is ready
Update local cache
Display data
from remote
callback
data from remote
is ready
Ignore data
from local cache
Main thread timeline

异步任务与队列

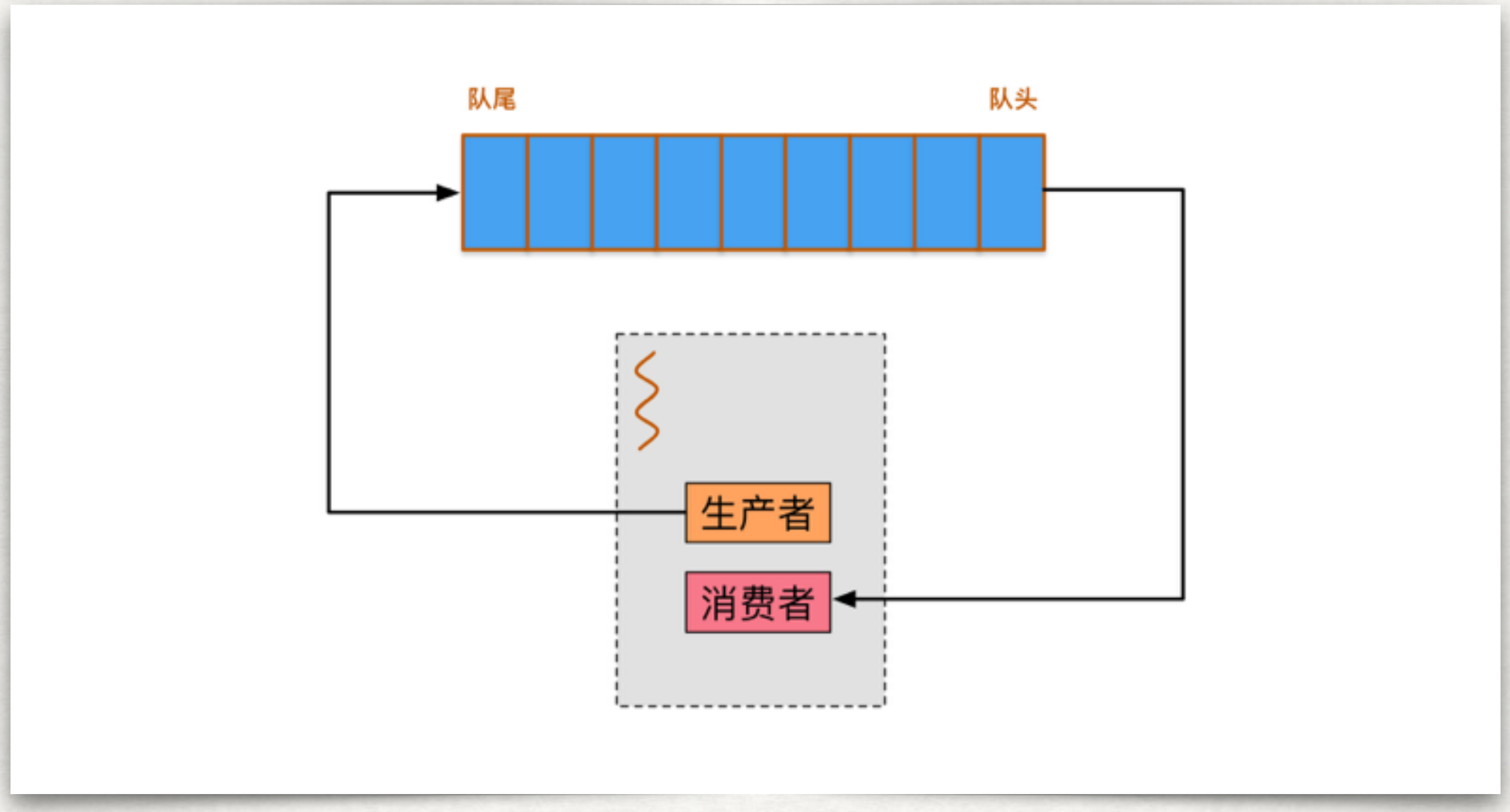
- 队列的应用场景
  - 发送聊天消息 (IM)
  - 上传多张照片
  - 关键的高频操作
- 任务队列
  - Thread-Safe Queue (TSQ)
  - 基于Callback的任务队列 (无锁队列)

Thread-Safe Queue



- 生成者和消费者不在同一个线程
- 任务是同步的

基于Callback的任务队列



- 生成者和消费者在同一个线程
- 任务是异步的

基于Callback的任务队列接口定义

```java
public interface TaskQueue {
    /**
     * Add new task to queue.
     * @param task
     */
    void addTask(Task task);

    /**
     * Set listener for callback.
     * @param listener
     */
    void setListener(TaskQueueListener listener);
}
```

```java
public interface TaskQueueListener {
    /**
     * Called when the task
     * completes successfully.
     * @param task
     */
    void taskComplete(Task task);
    /**
     * Called when the task finally fails
     * due to retrying too many times.
     * @param task
     * @param cause the cause of failure.
     */
    void taskFailed(Task task, Throwable cause);
}
```

异步任务接口定义

```java
public interface Task {
    /**
     * Launch the asynchronous task.
     */
    void start();

    /**
     * Set listener for callback.
     * @param listener
     */
    void setListener(TaskListener listener);

}
```

```java
interface TaskListener {
    /**
     * Called when the task
     * completes successfully.
     * @param task
     */
    void taskComplete(Task task);
    /**
     * Called when the task fails.
     * @param task
     * @param cause the cause of failure.
     */
    void taskFailed(Task task, Throwable cause);
}
```

任务队列实现举例

```java
@Override
public void addTask(Task task) {
    // Add new task to queue.
    taskQueue.offer(task);
    task.setListener(this);

    if (taskQueue.size() == 1) {
        // Current task is the first one,
        // launch it immediately.
        launchNextTask();
    }
}
```

```java
private void launchNextTask() {
    // Peek the task from the head of queue,
    // but do not dequeue.
    Task task = taskQueue.peek();
    if (task == null) {
        //impossible case
        Log.e(TAG, "impossible: NO task in queue, unexpected!");
        return;
    }

    task.start();
    runCount = 1;
}
```

```java
/**
 * Routine after a task is finished (succeeds or
 * finally fails).
 * @param task
 * @param error
 */
private void finishTask(Task task, Throwable error) {
    // Callback
    if (listener != null && !stopped) {
        try {
            if (error == null) {
                listener.taskComplete(task);
            }
            else {
                listener.taskFailed(task, error);
            }
        }
        catch (Throwable e) {
            Log.e(TAG, "", e);
        }
    }
    task.setListener(null);

    //Dequeue the task.
    taskQueue.poll();

    //Launch the next task in queue.
    if (taskQueue.size() > 0 && !stopped) {
        launchNextTask();
    }
}
```

异步任务的取消

- 取消一个任务容易吗?

```
ScheduledExecutorService executor =
        Executors.newSingleThreadScheduledExecutor();

ScheduledFuture<?> future =
        executor.schedule(new Runnable() {
    @Override
    public void run() {
        // ...
    }
}, delay, TimeUnit.SECONDS);

future.cancel(true);
```

无法保证任务取消成功

- 没有底层API保证一定能够成功取消任务
- java.util.concurrent.Future.cancel

cancel

```
boolean cancel(boolean mayInterruptIfRunning)
```

Attempts to cancel execution of this task. This attempt will fail if the task has already completed, has already been cancelled, or could not be cancelled for some other reason. If successful, and this task has not started when `cancel` is called, this task should never run. If the task has already started, then the `mayInterruptIfRunning` parameter determines whether the thread executing this task should be interrupted in an attempt to stop the task.

After this method returns, subsequent calls to `isDone()` will always return `true`. Subsequent calls to `isCancelled()` will always return `true` if this method returned `true`.

Parameters:

`mayInterruptIfRunning - true if the thread executing this task should be interrupted; otherwise, in-progress tasks are allowed to complete`

Returns:

`false if the task could not be cancelled, typically because it has already completed normally; true otherwise`

无法保证任务取消成功

- java.lang.Thread.interrupt

interrupt

```
public void interrupt()
```

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.

If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.

If this thread is blocked in an I/O operation upon an `InterruptibleChannel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `ClosedByInterruptException`.

If this thread is blocked in a `Selector` then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's `wakeup` method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

Interrupting a thread that is not alive need not have any effect.

Throws:

`SecurityException` - if the current thread cannot modify this thread

保证任务取消成功的场景

- 在聊天消息重发中的一个例子
  - (1) 网络状态变差
  - (2) 发送消息请求失败
  - (3) 调度一个异步任务，在延迟一段时间之后重发
    - 可能重发多次
  - (4) 网络状态变好
  - (5) 取消之前调度的异步任务，立即重发
    - 这里的取消需要很强的保证。为什么?

利用start ID取消任务

```
/**
 * When we need to retry another time.
 */
retryTaskCurrentStartId = generateNewStartId();
final long startId = retryTaskCurrentStartId;
retryTaskFuture = retryTaskScheduler.schedule(new Runnable() {
    @Override
    public void run() {
        mainHandler.post(new Runnable() {
            @Override
            public void run() {
                if (startId == retryTaskCurrentStartId) {
                    retryTaskCurrentStartId = 0;
                    retryTaskFuture = null;
                    task.start();
                }
            }
        });
    }
}, delay, TimeUnit.SECONDS);
```

- 是否任何异步任务都能使用这种方式来取消?
- 任务的等幂性

```
/**
 * When the network becomes good and
 * we need retry immediately.
 */
if (retryTaskFuture != null) {
    retryTaskCurrentStartId = 0;
    retryTaskFuture = null;
    retryTaskFuture.cancel(true);
    task.start();
}
```

RxJava对于异步接口设计的影响

```java
public interface TaskQueue {
    /**
     * Add new task to queue.
     *
     * @param task
     * @param <R> The result type of Task.
     * @return an Observable.
     */
    <R> Observable<R> addTask(Task<R> task);
}
```

```java
/**
 * Asynchronous Task Interface.
 *
 * @param <R> The result type of Task.
 */
public interface Task <R> {
    /**
     *
     * Launch the asynchronous task.
     *
     * @return an Observable.
     */
    Observable<R> start();
}
```

- 不再需要单独设计回调Listener

RxJava的优缺点

- 优点
  - 简化了接口设计
  - 消除了回调接口的设计
  - 用统一的方式来表达和处理异步任务
- 缺点
  - 对于接口的调用者简单，对于接口的实现者困难
  - 依赖subscribe去驱动它的上游开始运行
  - 调试时奇怪的调用栈
  - 现成的异步任务 —> Observable，不容易理解

Thanks!

我的微信ID

微信公众号