The Asynchronous Programming Patterns in Android Development

Charles Zhang | 张铁蕾

About Me

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

“

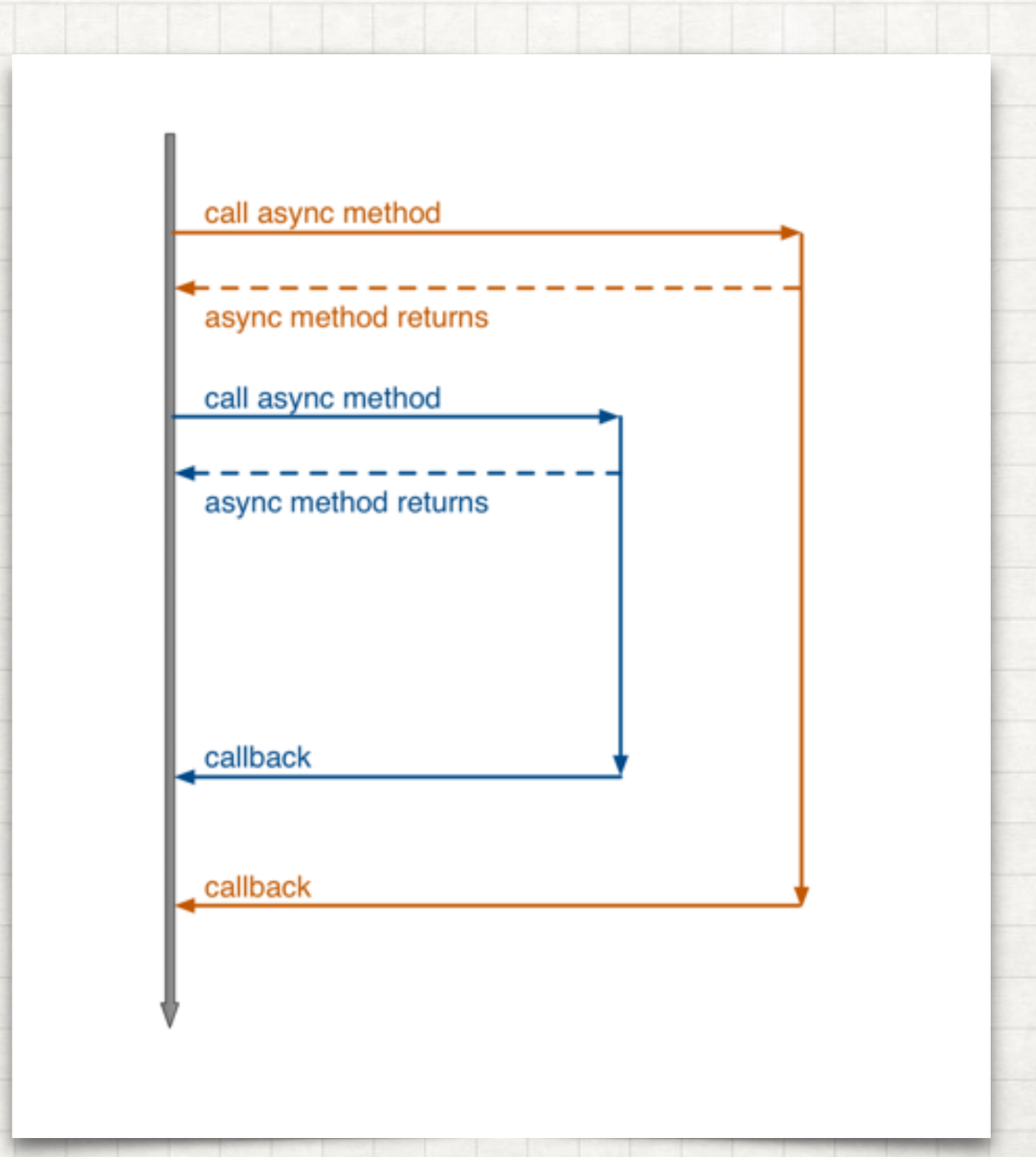
A common principle throughout computer science is that we can build complicated systems from minimal components.

— *Ian Goodfellow and Yoshua Bengio and Aaron Courville*

”

What is Async Programming?

- Async Tasks
  - Deviate from the main program flow
  - No Wait
- Async Events
  - Occur at any time.
  - e.g. BroadcastReceiver



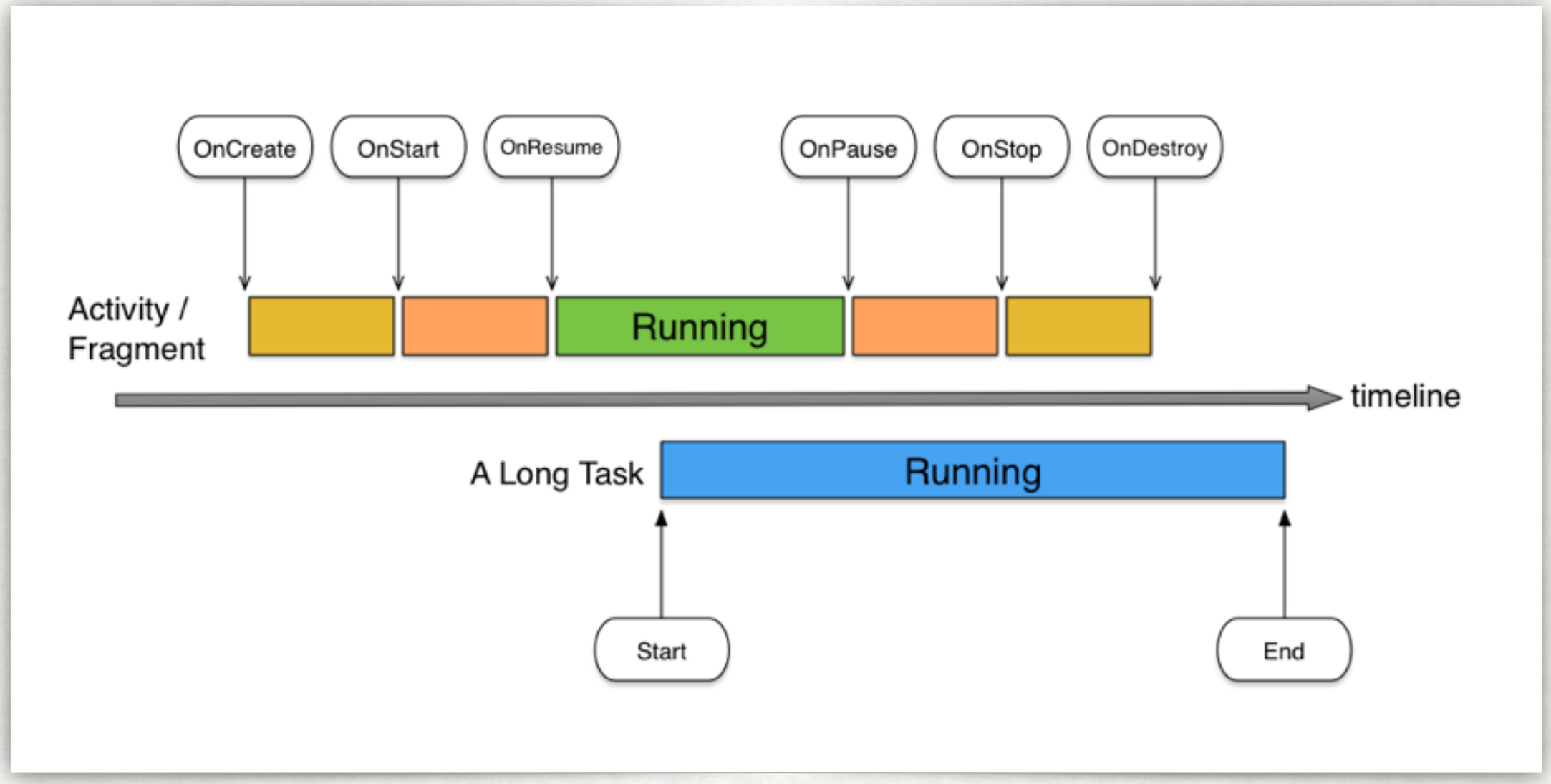
Async Programming in Android

- Frontend Programming (UI)
  - Layout
  - Input Events Processing
  - Drawing & Rendering
- Backend Programming (Logic)
  - Data Storage & Caching
  - Queueing
  - Networking
  - Lifecycle Management

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

Relevance of Async Programming to Frontend & Backend Programming

Example: Early Destroyed Activity/Fragment



- Examples of Long Task
  - Network Request, Downloader, Uploader, etc.

Example: Risks at the end of Long Task

- In Activity
  - findViewById(id) returns null.
- In Fragment
  - getActivity() returns null.
  - getView() returns null.
  - getResources().getString(id) throws IllegalStateException.

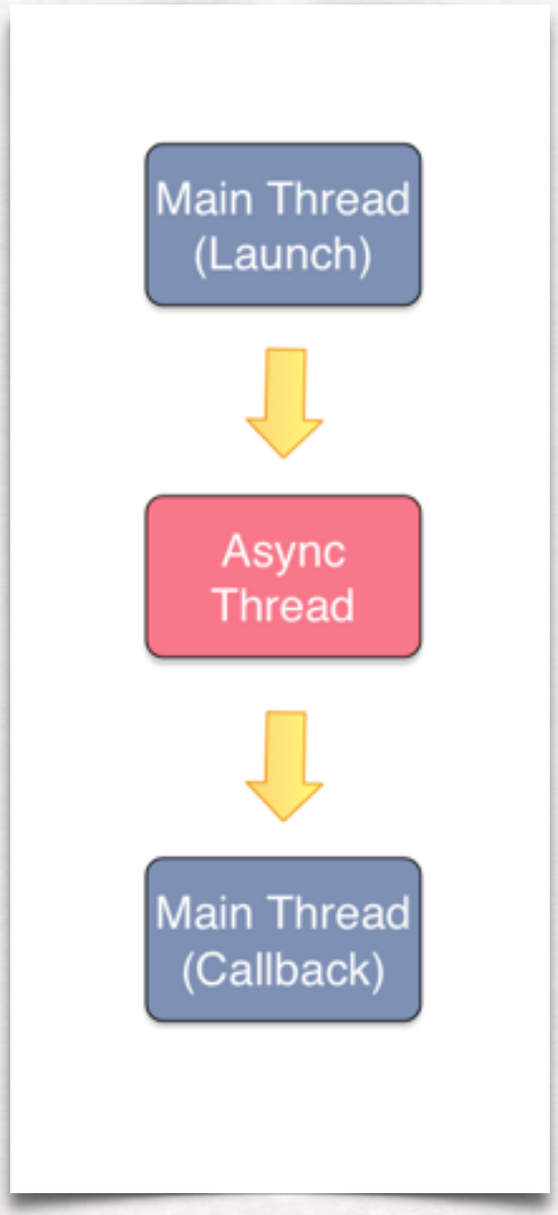
Example: Possible Solutions

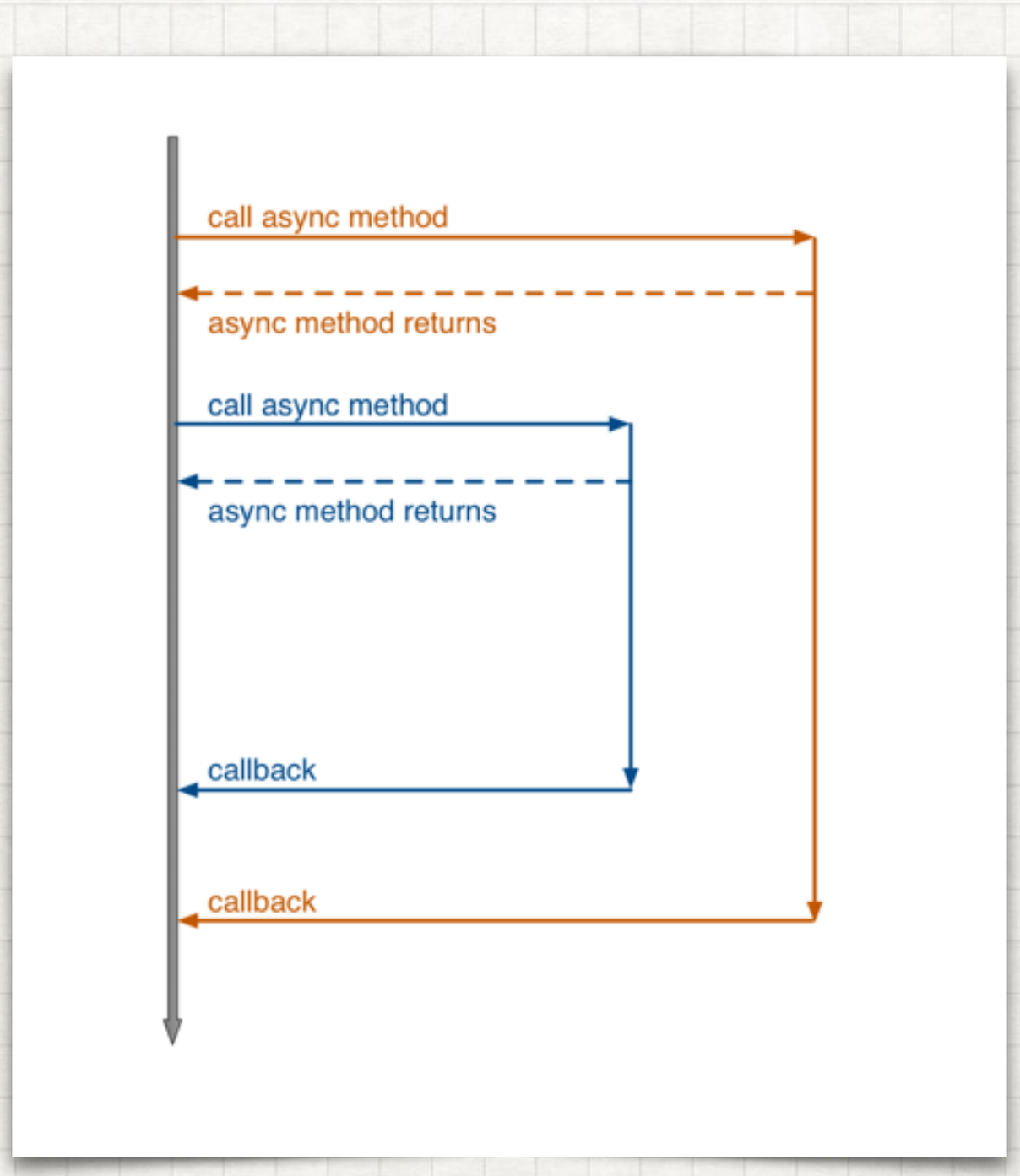
- Cancel
- Ignore
- Bind / Unbind

How to Launch an Async Task in Android?

- Network Requests
  - e.g. Volley, Retrofit
- Thread Pool Scheduling
  - e.g. ExecutorService, ScheduledExecutorService
- Looper Scheduling
  - e.g. Handler.post/sendMessage, View.postDelayed, Handler.postDelayed/postAtTime
- System Behaviors
  - e.g. startActivity, launching a fragment

General Threading Pattern of Async Task





CALLBACK TO MAIN THREAD!

Some Async Programming Patterns

I. Cooperation between Multiple Async Tasks

II. Queueing with Async Task

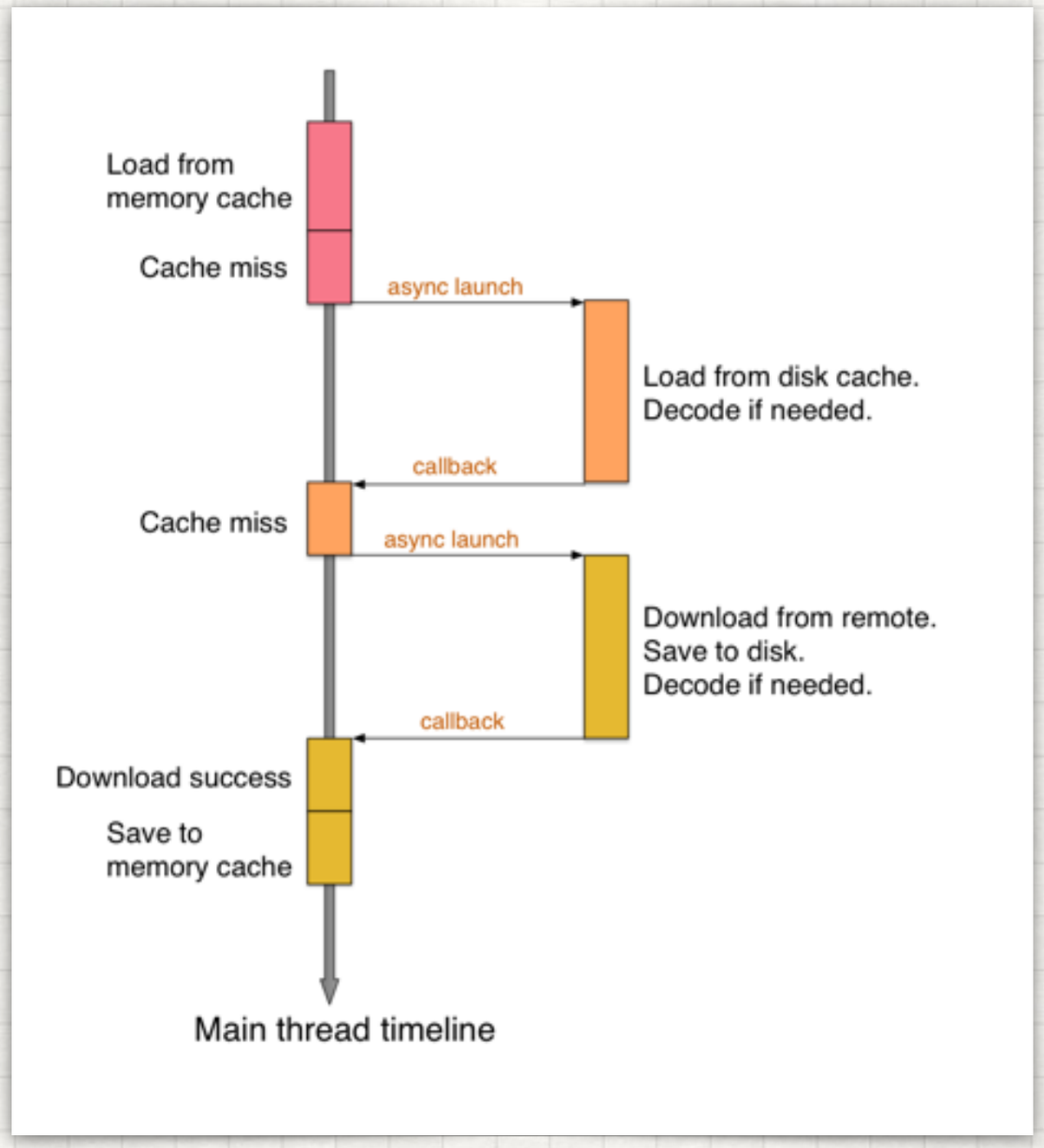
III. Cancellation of Async Task

Cooperation between Multiple Async Tasks

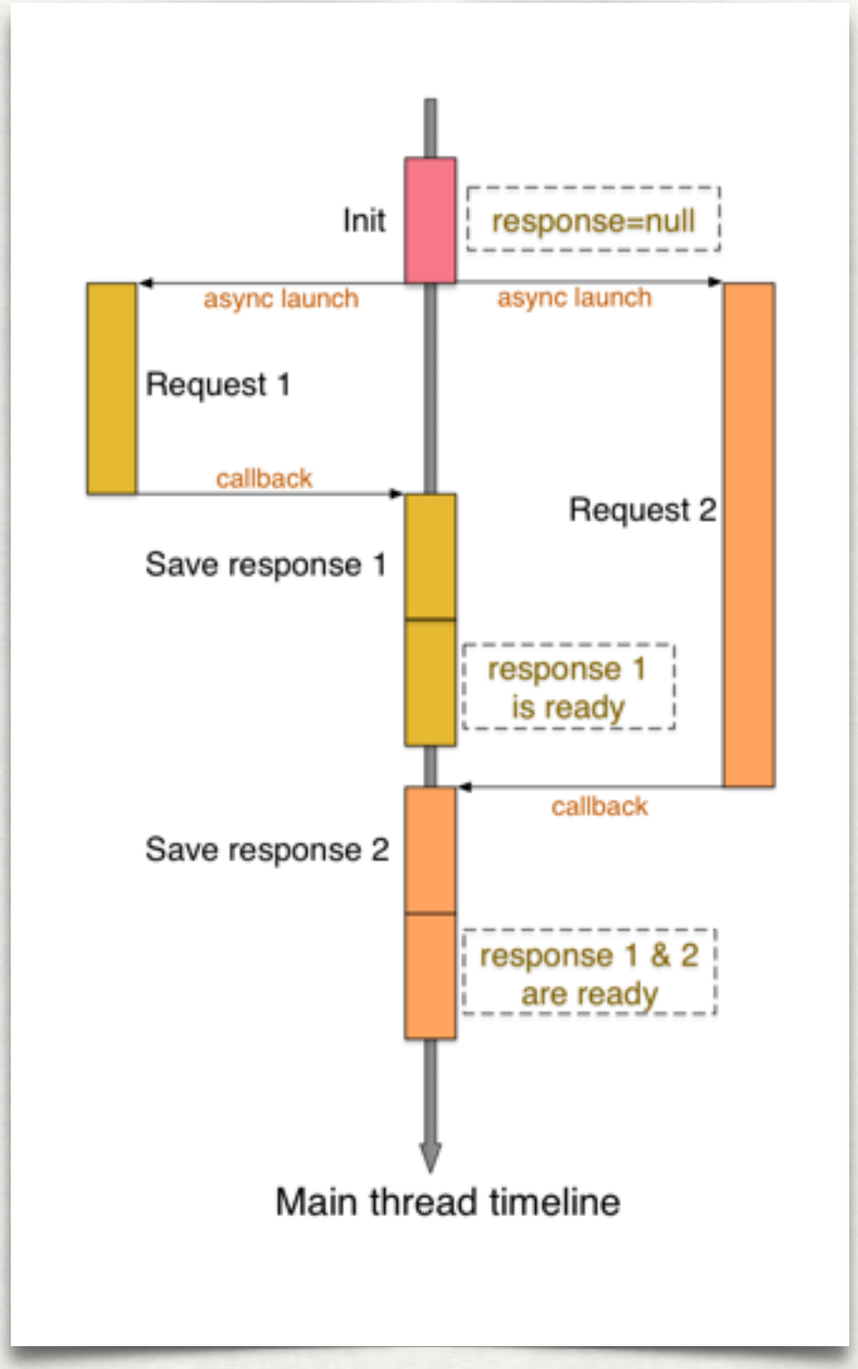
- Serial
  - e.g. Multi-Level Caching
- Merge
  - e.g. Simultaneous Network Requests
- Merge with Priority
  - e.g. Simple Page Caching

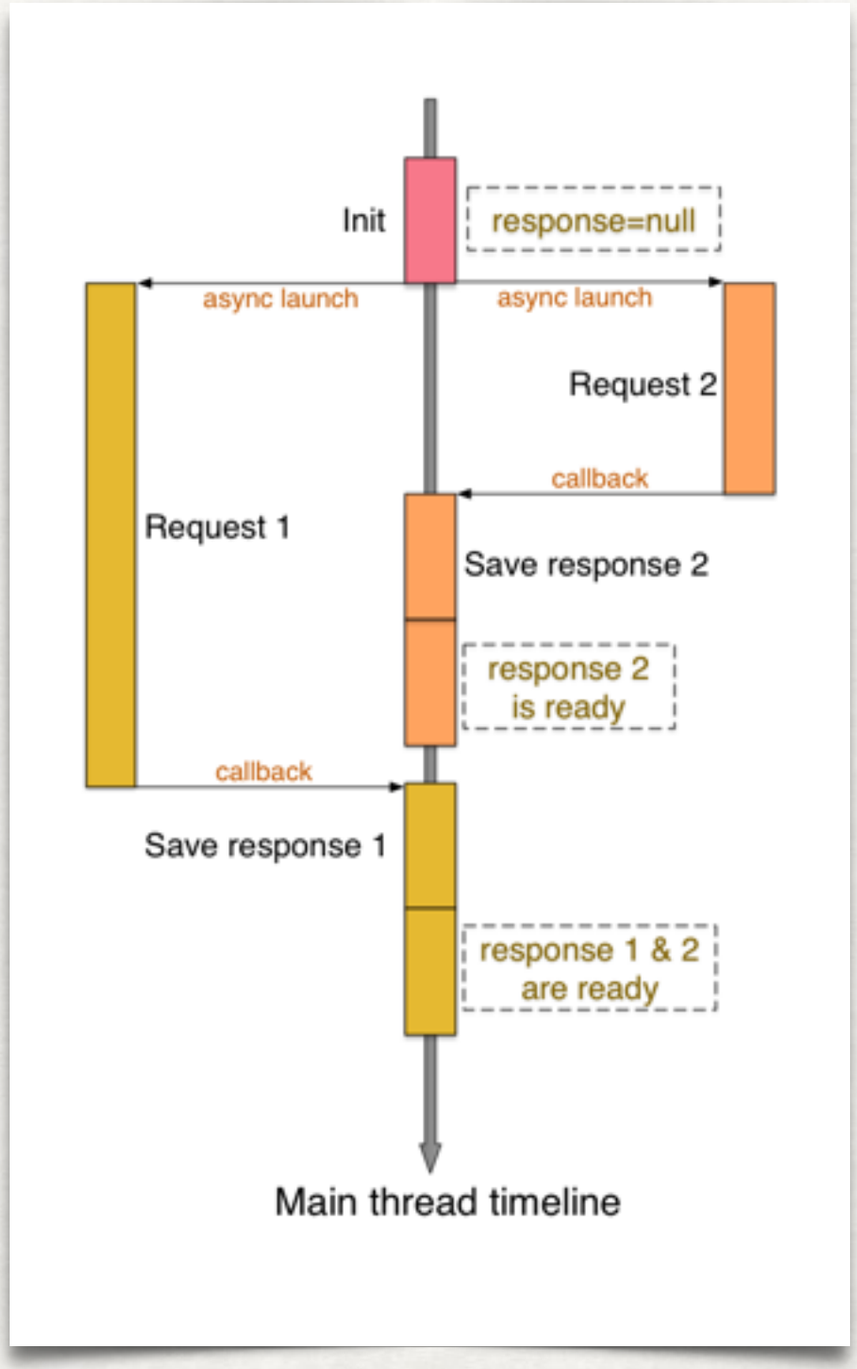
Multi-Level Caching / Serial

- Memory Cache (sync)
- Disk Cache (async)
- Download from Remote (async)



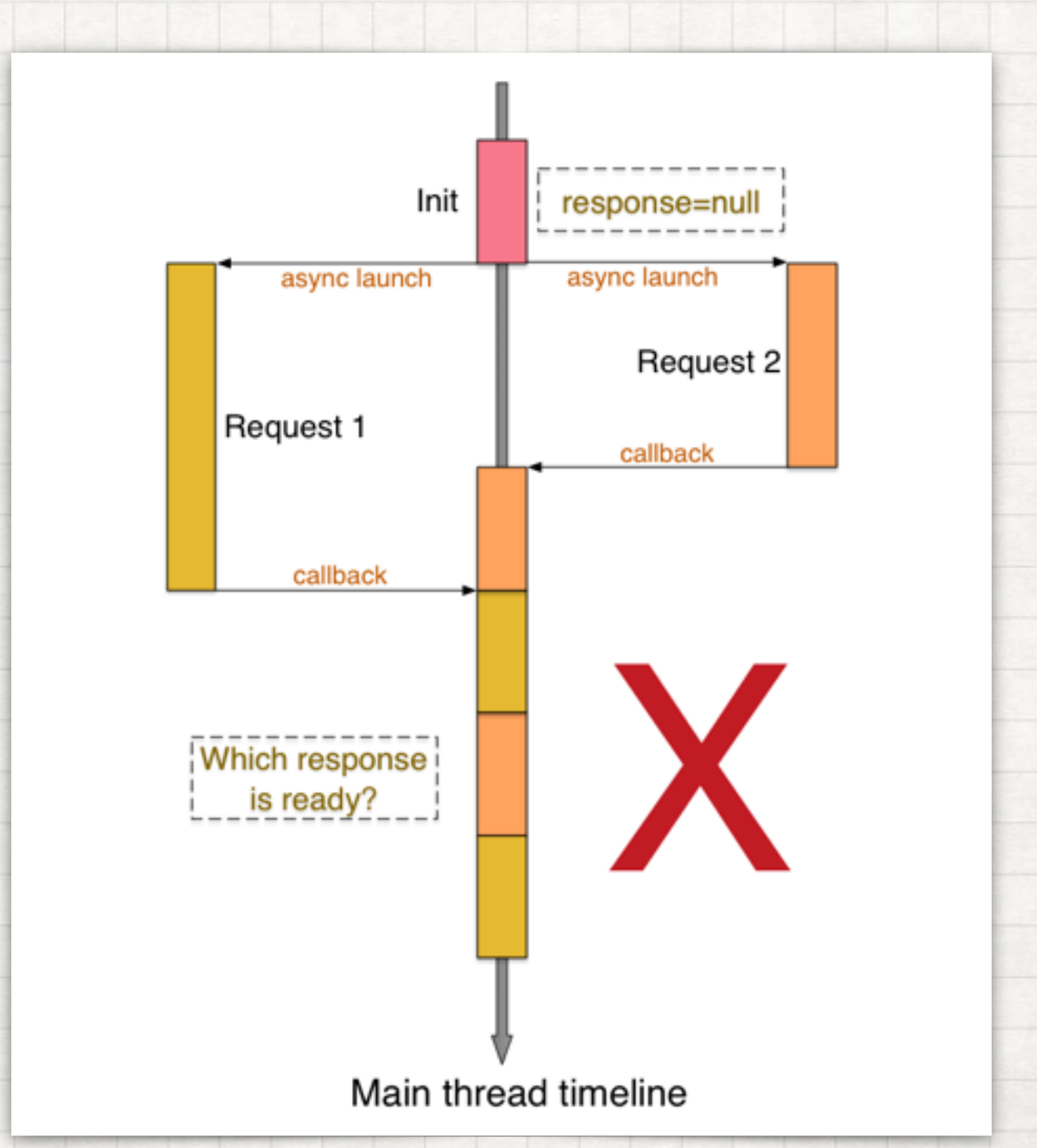
Simultaneous Network Requests / Merge



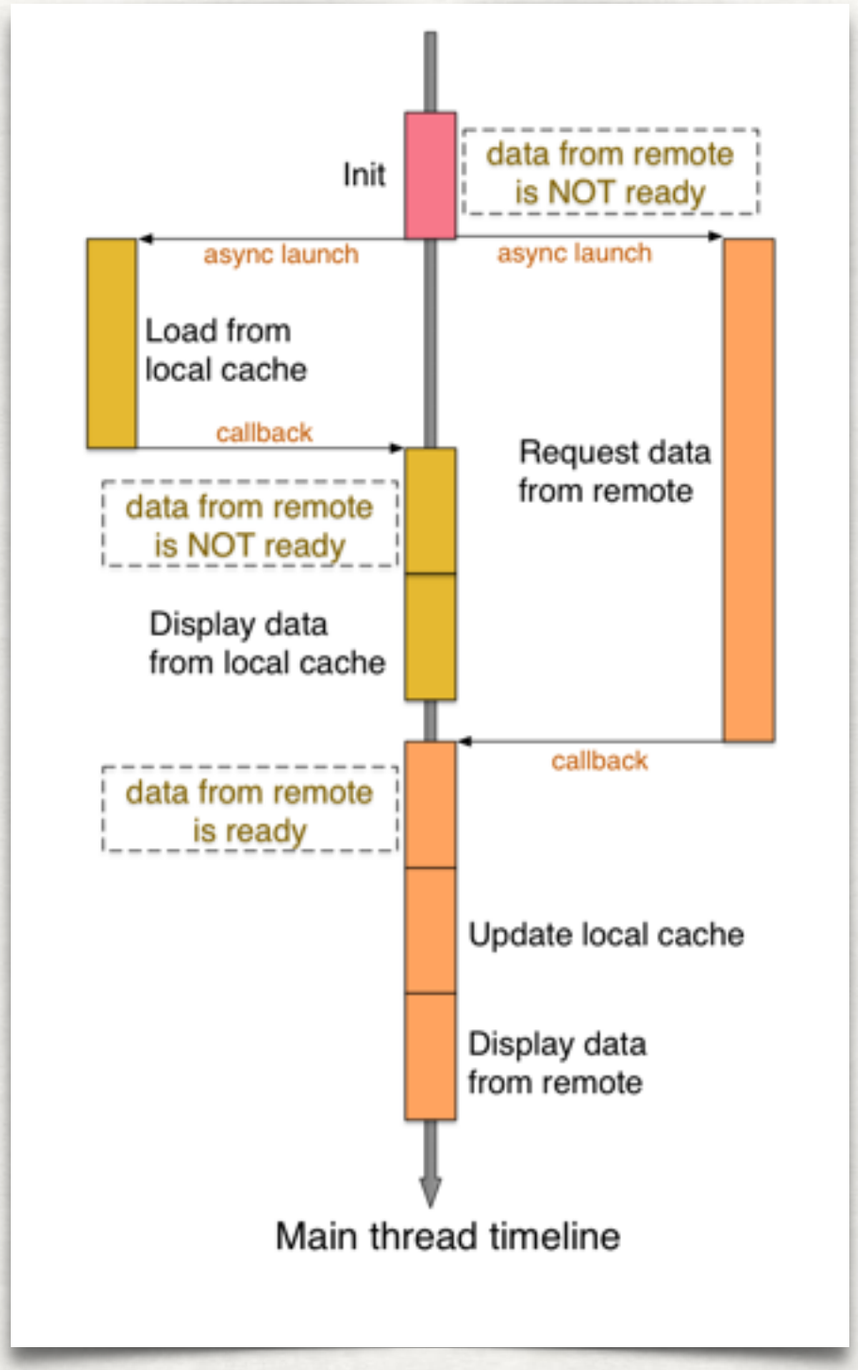


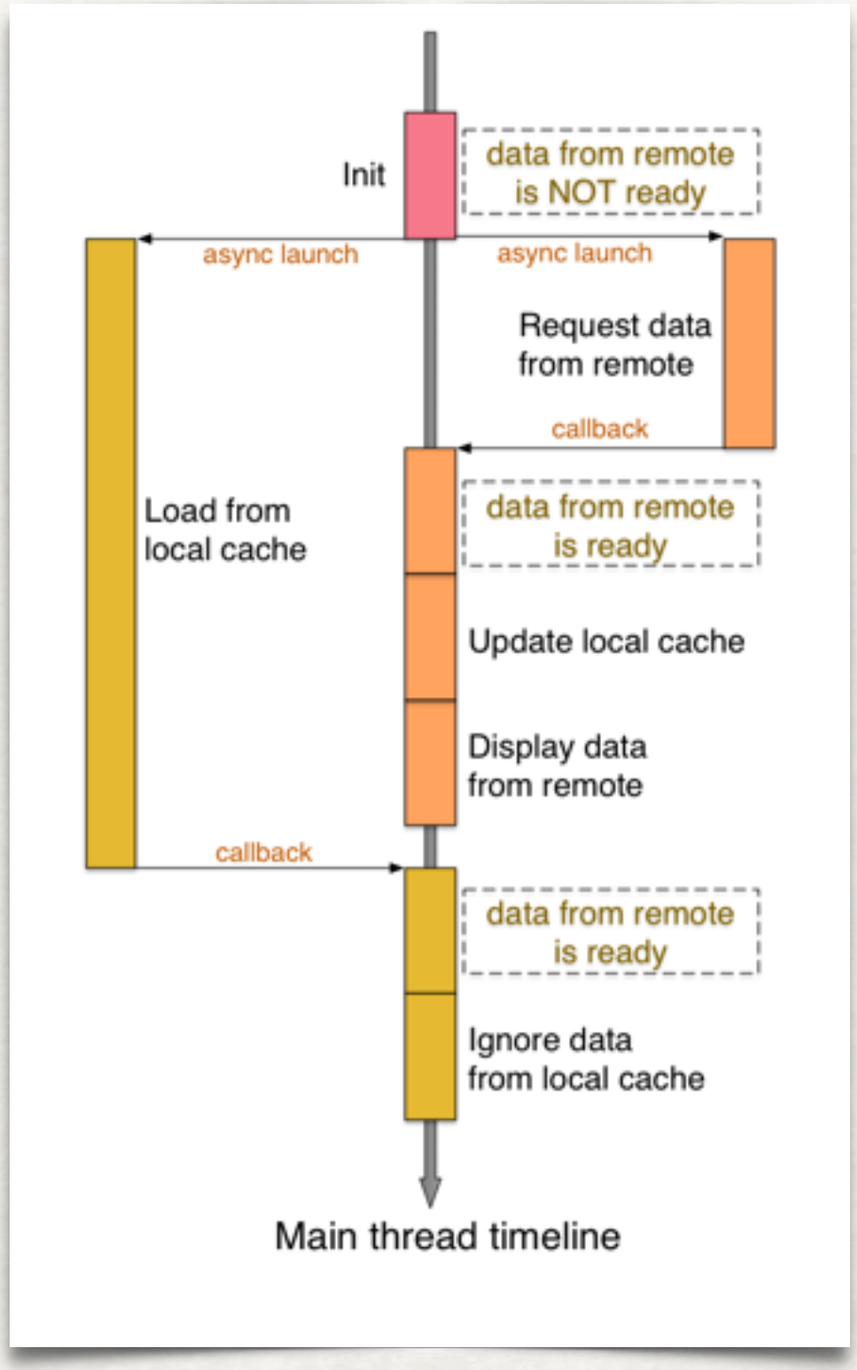
Simultaneous Network Requests / Merge

- It is NOT possible to interleave code on the same main thread!



Page Caching / Merge with Priority

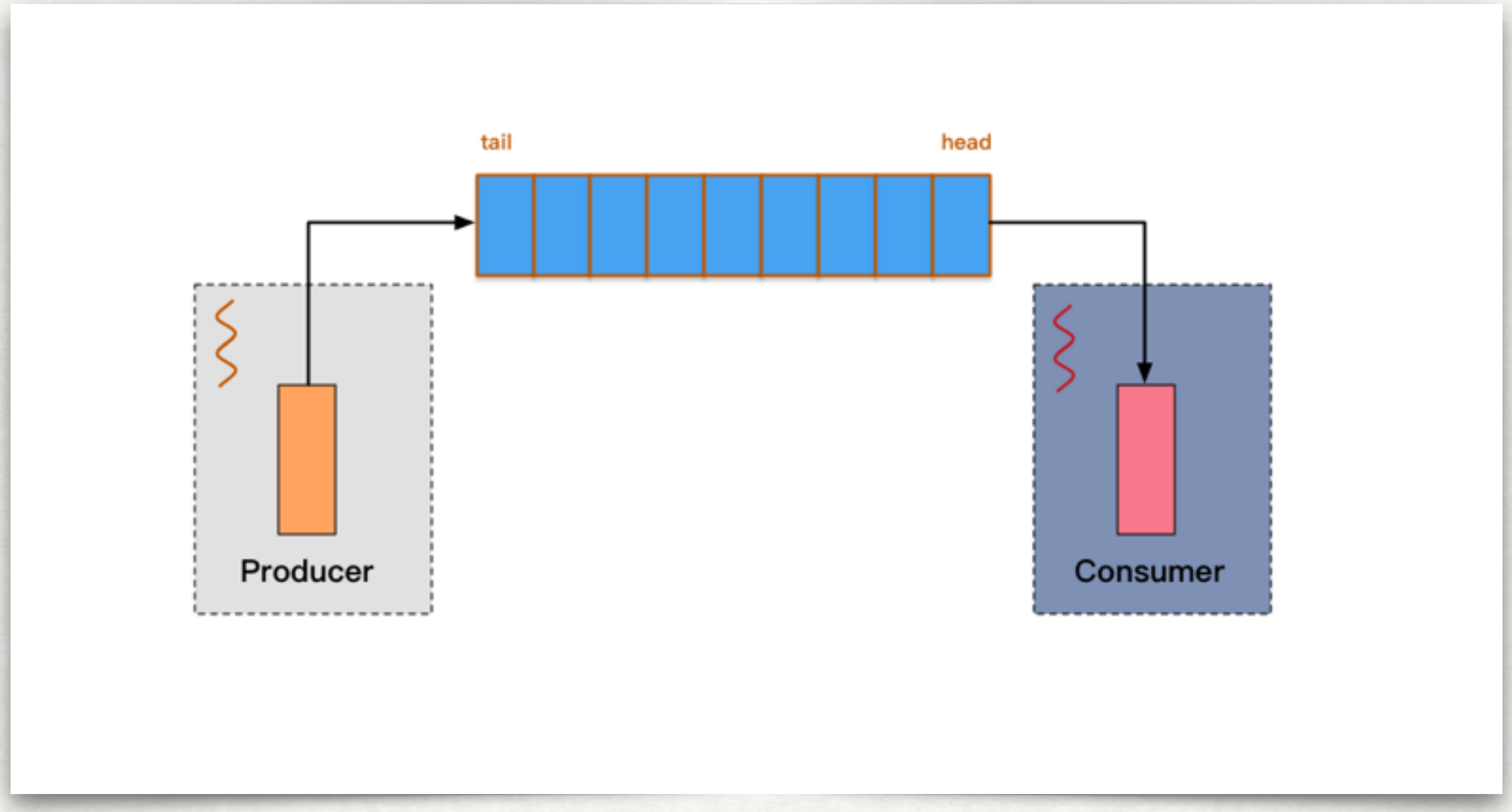




Queueing with Async Task

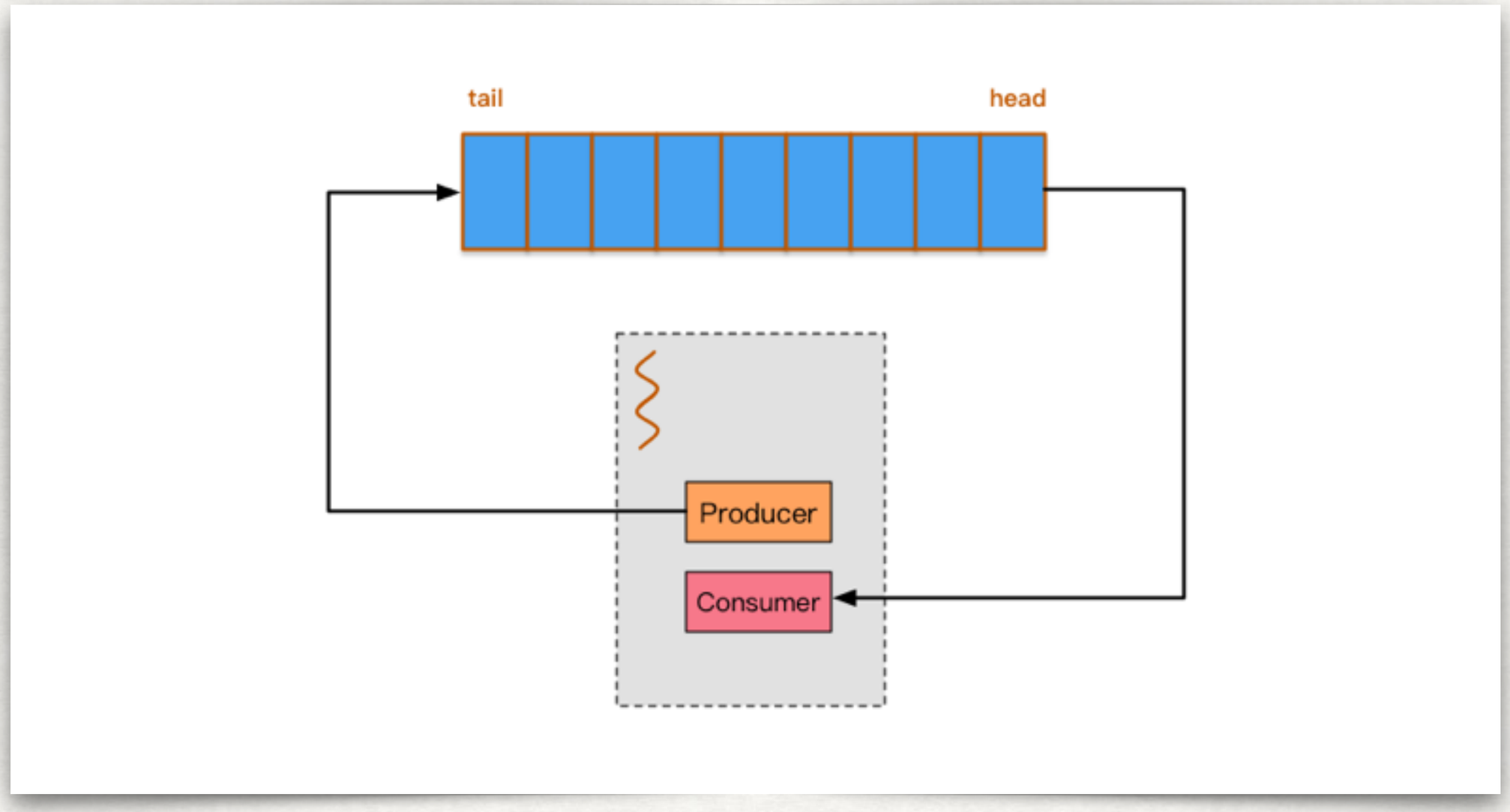
- Scenarios where a Queue is Needed
  - Chat Send Queue (in IM)
  - Upload Photos
  - Critical High Frequency Operations
- Task Queue
  - Thread-Safe Queue (TSQ)
  - Callback-Based Queue (Lock-Free Queue)

Thread-Safe Queue



- Producer Thread VS. Consumer Thread
- Task is Synchronous

Callback-Based Queue



- Producer & Consumer in the Same Thread
- Task is Asynchronous

Callback-Based Queue Interface

```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);
}
```

Asynchronous Task Interface

```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);
}
```

Task Queue Implementation Example

```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();
    }
}
```

Cancellation of Async Task

- Is it easy to cancel a task?

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

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

future.cancel(true);
```

Cancellation: NO Strong Guarantees

- NO underlying API guarantees cancellation.
- 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`

Cancellation: NO Strong Guarantees

- 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

Strong Guarantee Requirements for Cancellation

- An Example in Chat Message Resending
  - (1) Network status becomes bad.
  - (2) Message request fails.
  - (3) Schedule with delay to resend message.
    - May resend several times.
  - (4) Network status becomes good.
  - (5) Cancel the delayed schedule and resend immediately.
    - This cancellation requires strong guarantees. Why?

Cancellation with start ID

```
/**
 * 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);
```

- Is any async task able to be cancelled in this way?
- Idempotent task.

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

Effect of RxJava on Async Interface Design

```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();
}
```

- Callback listeners are no longer needed.

Pros and Cons of RxJava

- Pros
  - Interface design is simplified
  - Callback listeners are eliminated
  - Unified representation & processing of async task
- Cons
  - Easy to callers, difficult to implementors
  - Dependence on subscribe
  - Strange stacks when debugging
  - Existing async task —> Observable, not easy to understand

Thanks!

My WeChat ID

WeChat公众号