



# План занятия

1. [Задача](#)
2. [Thread Pools](#)
3. [Async](#)
4. [Итоги](#)
5. [Домашнее задание](#)



# ЗАДАЧА



# ВВЕДЕНИЕ

На прошлой лекции мы остановились на том, что возможности создавать вручную потоки и использовать механизмы синхронизации вроде `synchronized` или `Atomic`'ов недостаточно.

---

## ПРИМЕР

Речь идёт о следующей ситуации: мы загружаем список постов и для каждого поста нужно загрузить доп.контент:

- аватарку автора
- attachments (если они есть)
- комментарии\*

Примечание\*: вспомните, в том же Vk в объекте поста нет комментариев, для их получения нужен отдельный запрос.

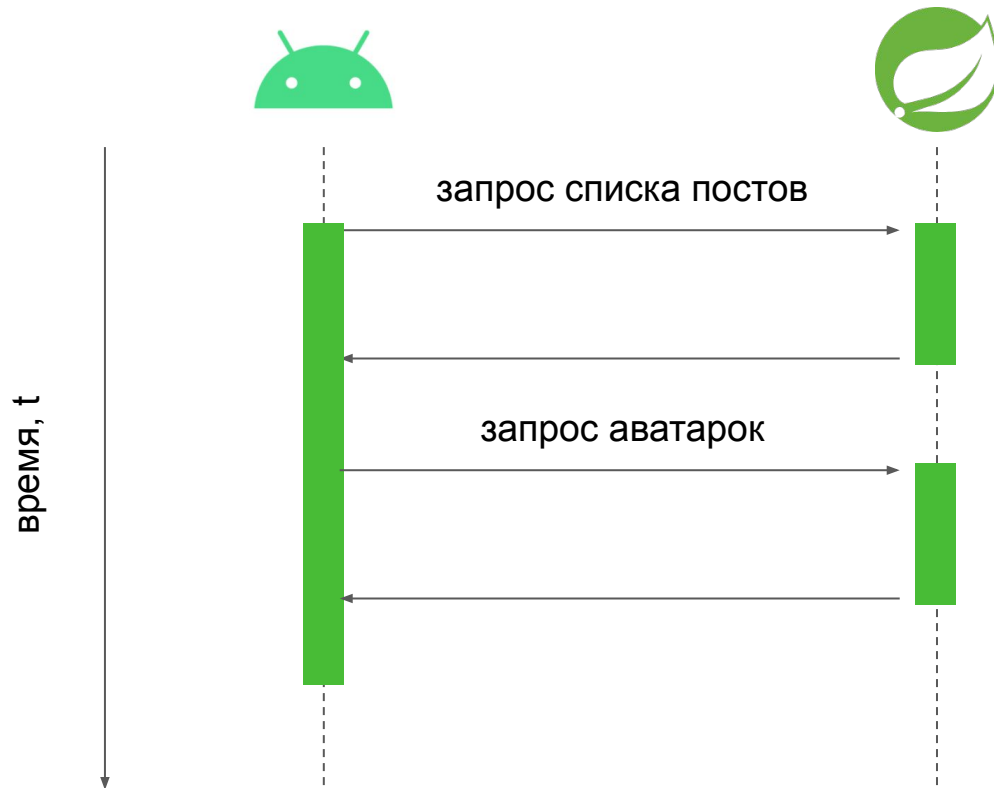


## ЗАМЕЧАНИЕ

Для вас, как для мобильного разработчика, идеальным сценарием будет использование одного вызова API для получения всей информации, необходимой для отрисовки экрана.

Не всегда так бывает, но вы вполне можете попробовать договориться об этом с разработчиком API.

# ПРИМЕР



Обратите внимание: когда мы смотрим на эту картинку, мы не видим никаких потоков, `volatile`, `synchronized`, `atomics` и `thread safe` коллекций. **Мы видим задачи.**

---

# CONCURRENCY

?

*Вопрос: как вы думаете, сколько запросов нужно сделать, чтобы загрузить аватарки авторов 10 постов?*

Примечание: иногда на собеседовании любят задавать такие вопросы на «логику», проверяя, как вы думаете и видите ли пути оптимизации.

---

# IMAGES

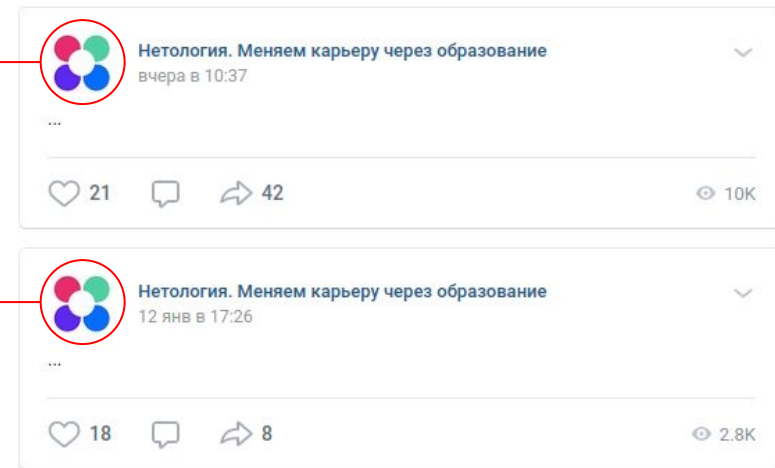
В простейшем случае изображения хранятся в виде обычных файлов и на каждый файл вам нужно будет делать отдельный HTTP-запрос.




# IMAGES

Возможные оптимизации:


1. Если подряд идут посты одного автора, не нужно два раза скачивать одно и то же изображение



2. Можно закешировать изображение, сохранив его в файловой системе



Clear storage



Clear cache

---

SPACE USED

App size

59.55 MB

User data

40.39 MB

Cache

8.35 MB



# THREAD POOLS



# THREADS

Запускать большое количество конкурентных потоков — плохая идея из-за большого расхода ресурсов.

Можно ли как-то переиспользовать потоки?



# THREAD POOLS

«Готовые» пулы потоков, предназначенные для выполнения конкретной задачи или ряда задач.

# EXECUTOR SERVICE

«Исполнители» задач:

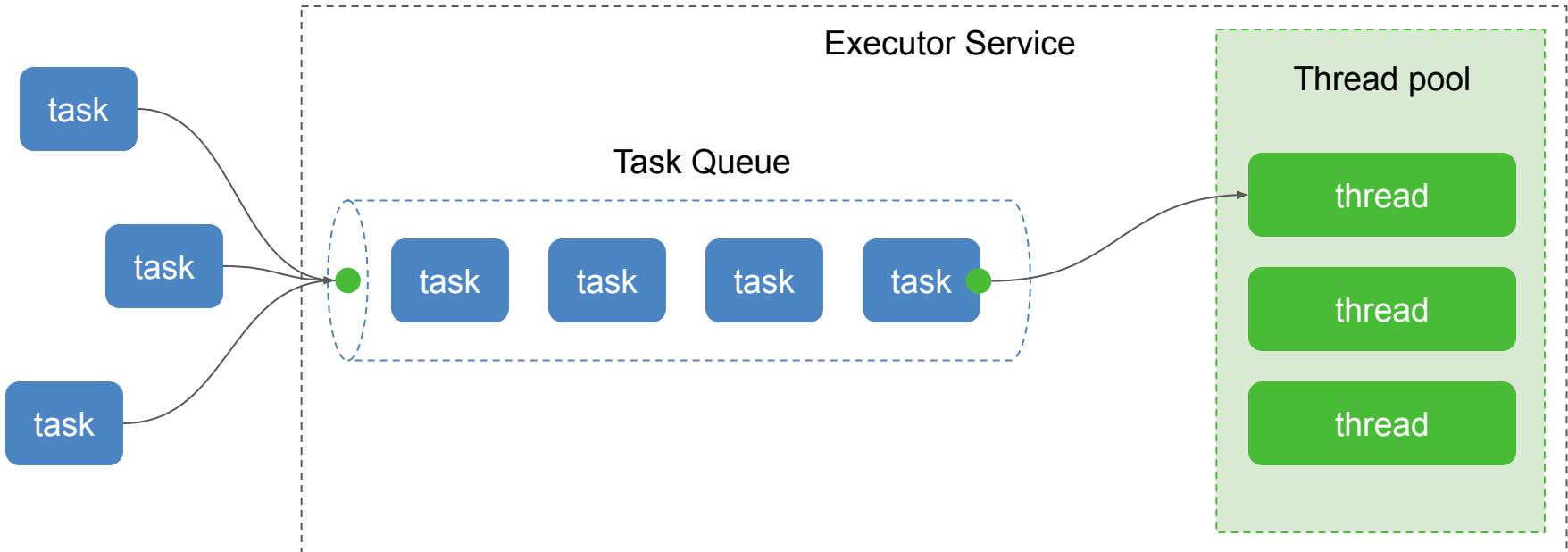
- сами распределяют задачи
- сами управляют очередью задач
- «работают» с потоками

```
public interface ExecutorService extends Executor {  
    @NotNull  
    <T> Future<T> submit( @NotNull Callable<T> task);  
  
    @NotNull  
    Future<?> submit( @NotNull Runnable task);  
}
```

Callable — аналог «Runnable», но с возвратом результата.

Future — абстракция результата будущих вычислений.

# EXECUTOR SERVICE\*



Для этого стандартная библиотека Java предлагает пулы потоков (наборы готовых потоков, на которых можно исполнять задачи).

Примечание\*: схема «типичных» Executor Service (см. следующий слайд).

# EXECUTORS

Ключевые готовые реализации:

- **FixedThreadPool** — сервис, использующий пул потоков фиксированного размера (при тонкой настройке можно задавать политику минимального количества потоков)
- **CachedThreadPoolExecutor** — сервис, использующий динамический пул потоков (переиспользуя существующие потоки, либо создавая новые, если необходимо)
- **ForkJoinPool** (WorkStealingPool) - сервис, использующий механизм Work Stealing'a (для вычислительных задач, хорошо поддающихся разбиению на части)

# EXECUTORS

```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
                                   keepAliveTime: 0L, TimeUnit.MILLISECONDS,  
                                   new LinkedBlockingQueue<Runnable>());  
}
```

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(corePoolSize: 0, Integer.MAX_VALUE,  
                                   keepAliveTime: 60L, TimeUnit.SECONDS,  
                                   new SynchronousQueue<Runnable>());  
}
```



```

val postsRequest = Request.Builder().url(url: "$baseUrl/api/posts").build()
val posts: List<Post> = client.newCall(postsRequest)
    .execute()
    .let { it.body?.string() ?: throw RuntimeException("body is null") }
    .let { gson.fromJson(it, typeToken.type) }

val avatarsLoadTasks = posts.asSequence()
    .map { it.authorAvatar }
    .distinct()
    .map { it to Request.Builder().url(url: "$baseUrl/avatars/$it").build() }
    .map { (url, request) ->
        Callable {
            println("executed in ${Thread.currentThread().name}")
            client.newCall(request)
                .execute()
                .body?.use { it: ResponseBody
                    Files.copy(it.byteStream(), Paths.get(url), StandardCopyOption.REPLACE_EXISTING)
                    url ^use
                } ^Callable
            }
    }
    }.toMutableList()

val newFixedThreadPool = Executors.newFixedThreadPool(nThreads: 64)
val list = newFixedThreadPool.invokeAll(avatarsLoadTasks)
for (future in list) {
    println(future.get()) ← blocking call
}
newFixedThreadPool.shutdown()

```

---

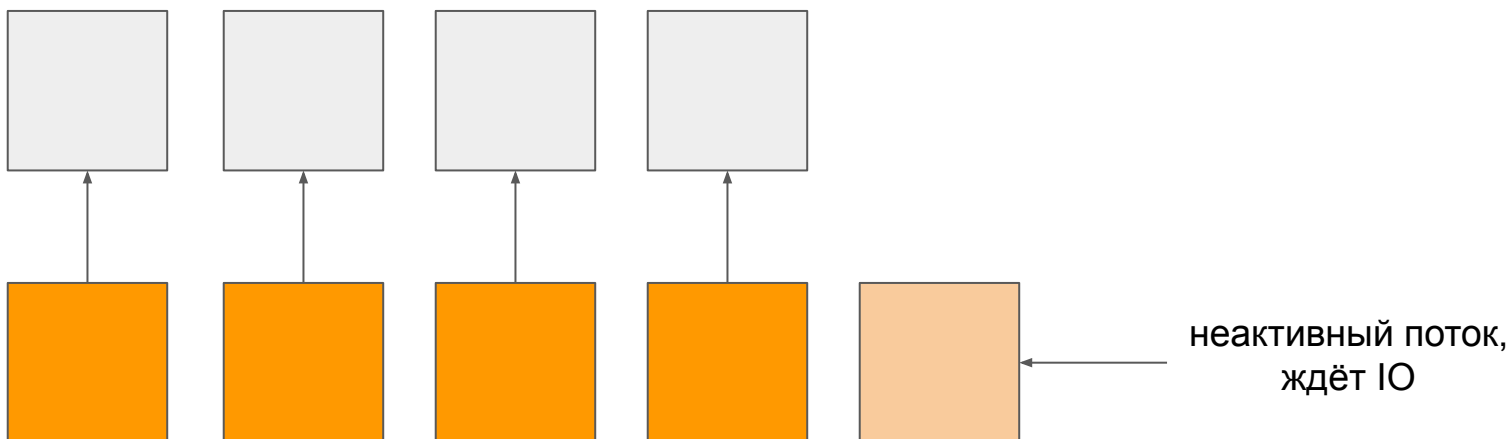
## ОСНОВНАЯ ИДЕЯ

**Q:** Почему мы указали 64 потока, у нас же явно нет столько ядер?

**A:** Дело в том, что когда поток блокируется в ожидании завершения IO операций, ядро процессора может переключиться на другой поток (которому не надо ждать). Поэтому даже если у нас 4-8 ядер, мы вполне себе для IO операций можем создать и 64 потока.

# ОСНОВНАЯ ИДЕЯ

ядра процессора  
( $P = 4$ )



активные потоки  
( $T = 4$ )

# СКОЛЬКО НУЖНО ПОТОКОВ?

Всё зависит от задачи:

- для CPU задач = количеству ядер:

`Runtime.getRuntime().availableProcessors()`

- для IO задач = ограничениям с серверной стороны или (в общем случае):

$\text{Number of Available Cores} * (1 + \text{Wait time} / \text{Service time})^*$

Примечание\*: из книги Java Concurrency in Practice



## ЗАЧЕМ ЭТО ВСЁ?

Вы должны понять идею Thread Pool'ов и идею разделения задач на CPU vs IO. Эта идея будет проходить через всю вашу дальнейшую разработку в Android.



## ОКНТТР

На самом деле, не обязательно создавать свой пул, потому что большинство библиотек уже используют под капотом свой пул, предлагая асинхронные методы вызова.



**ASync**

---

# ASYNC

**Q:** Что такое асинхронные вызовы?

**A:** По факту — это вызовы функций, в которые мы прокидываем callback'и, которые будут вызваны тогда, когда произойдёт нужное нам событие (например, придёт ответ на HTTP-запрос).





# ASYNC

Для начала попробуем решить задачу использования пула потоков вместо запуска нового потока на каждый HTTP запрос (к загрузке изображений вернёмся на следующей лекции).

# OKHTTP

```
fun enqueue(responseCallback: Callback)
```



```
interface Callback {
```

Called when the request could not be executed due to cancellation, a connectivity problem or timeout. Because networks can fail during an exchange, it is possible that the remote server accepted the request before the failure.

```
fun onFailure(call: Call, e: IOException)
```

Called when the HTTP response was successfully returned by the remote server. The callback may proceed to read the response body with [Response.body](#). The response is still live until its response body is [closed](#). The recipient of the callback may consume the response body on another thread.

Note that transport-layer success (receiving a HTTP response code, headers and body) does not necessarily indicate application-layer success: response may still indicate an unhappy HTTP response code like 404 or 500.

```
@Throws(IOException::class)
```

```
fun onResponse(call: Call, response: Response)
```

```
}
```

# OKHTTP

```
fun getAllAsync() {  
    val request: Request = Request.Builder()  
        .url(url: "${BASE_URL}/api/slow/posts")  
        .build()  
  
    client.newCall(request)  
        .enqueue(object : Callback {  
            override fun onResponse(call: Call, response: Response) {  
                TODO(reason: "Not yet implemented")  
            }  
  
            override fun onFailure(call: Call, e: IOException) {  
                TODO(reason: "Not yet implemented")  
            }  
        })  
}
```

# GETALL

Теперь возникает вопрос: если раньше мы просто возвращали данные из метода, то как выходить из ситуации сейчас?

```
override fun getAll(): List<Post> {  
    val request: Request = Request.Builder()  
        .url(url: "${BASE_URL}/api/slow/posts")  
        .build()  
  
    return client.newCall(request)  
        .execute()  
        .let { it.body?.string() ?: throw RuntimeException("body is null") }  
        .let { it: String?  
            gson.fromJson(it, typeToken.type)  
        }  
}
```

---

# REPOSITORY

```
interface PostRepository {  
    fun getAll(): List<Post>  
    fun likeById(id: Long)  
    fun save(post: Post)  
    fun removeById(id: Long)  
  
    fun getAllAsync(callback: GetAllCallback)  
  
    interface GetAllCallback {  
        fun onSuccess(posts: List<Post>) {}  
        fun onError(e: Exception) {}  
    }  
}
```

# GETALLASYNC

```
override fun getAllAsync(callback: PostRepository.GetAllCallback) {  
    val request: Request = Request.Builder()  
        .url(url: "${BASE_URL}/api/slow/posts")  
        .build()  
  
    client.newCall(request)  
        .enqueue(object : Callback {  
            override fun onResponse(call: Call, response: Response) {  
                val body = response.body?.string() ?: throw RuntimeException("body is null")  
                try {  
                    callback.onSuccess(gson.fromJson(body, typeToken.type))  
                } catch (e: Exception) {  
                    callback.onError(e)  
                }  
            }  
  
            override fun onFailure(call: Call, e: IOException) {  
                callback.onError(e)  
            }  
        })  
}
```

# VIEWMODEL

```
fun loadPosts() {  
    _data.value = FeedModel(loading = true)  
    repository.getAllAsync(object : PostRepository.GetAllCallback {  
        override fun onSuccess(posts: List<Post>) {  
            _data.postValue(FeedModel(posts = posts, empty = posts.isEmpty()))  
        }  
  
        override fun onError(e: Exception) {  
            _data.postValue(FeedModel(error = true))  
        }  
    })  
}
```



# ИТОГИ





## ИТОГИ

Сегодня мы обсудили пулы потоков и их применимость. Это достаточно важно, поскольку далее (в корутинах) будет использоваться на каждом шагу.



## ИТОГИ

На следующей лекции мы посмотрим, какие инструменты есть в самом Android, чтобы решить нашу задачу с загрузкой изображений, а также познакомимся с библиотекой Glide, которая и будет решать большую часть вопросов с загрузкой изображений.