数据持久化就是将数据保存在计算机的硬盘中，这样无论是在断网还是重启计算机时，都可以访问之前保存的数据。在 iOS 开发中，有以下 4 种数据持久化的方案。

- **Plist**。它是一个 XML 文件，会将某些固定类型的数据存放于其中，读写分别通过 contentsOfFile 和 writeToFile 来完成。Plist 一般用于保存 App 的基本参数。

- **Preference**。它通过 UserDefaults 来完成 key-value 配对保存。如果需要立刻保存，则需要调用 synchronize 方法。它会将相关数据保存在同一个 plist 文件下。Preference 同样用于保存 App 的基本参数信息。

- **NSKeyedArchiver**。遵循 NSCoding 协议的对象就可以实现序列化。NSCoding 有两个必须要实现的方法，即父类的归档（initWithCoder 方法）和解档（encodeWithCoder 方法）。存储数据通过 NSKeyedArchiver 的工厂方法 archiveRootObject:toFile:来实现；读取数据通过 NSKeyedUnarchiver 的工厂方法 unarchiveObjectwithFile:来实现。相比前两者，NSKeyedArchiver 可以任意指定文件存储的位置和文件名。

- **CoreData**。前面几种方法都是覆盖存储，在修改数据时要读取整个文件，修改后再覆盖写入，十分不适合大量数据的存储。CoreData 就是苹果公司官方推出的大规模数据持久化的方案。它的基本逻辑类似 SQL 数据库，每个表为 Entity，可以添加、读取、修改、删除对象实例。它可以像 SQL 一样提供模糊搜索、过滤搜索、表关联等各种复杂操作。虽然 CoreData 功能强大，但也有缺点，即学习曲线高，操作复杂。

以上 4 种方法是在 iOS 开发中最为常见的数据持久化方案。除这些外，针对大规模数据持久化，还可以用 SQLite 3、FMDB、Realm 等方法。相比 CoreData 和其他方案，Realm 凭借简便的操作和丰富的功能受到很多开发者的青睐。同时，一些大公司，诸如 Google 的 Firebase，也有离线数据库功能。其实没有最佳的方案，只有最合适的方案，应该根据实际开发的 App 来挑选合适的持久化方案。

4.4 并发编程

所有的语言都会涉及并发编程。并发就是多个任务同时运行，这也几乎是所有语言最难学

习的地方。在 iOS 开发中，并发编程主要用于提升 App 的运行性能，保证 App 实时响应用户的操作。我们日常操作的 UI 界面就是运行在主线程之上的，它是一个串行线程。如果将所有代码都放在主线程上运行，那么主线程将承担网络请求、数据处理、图像渲染等各种操作，无论是 GPU 还是计算机内存，都会性能耗尽，从而影响用户体验。

本节会从并发编程的理论说起，重点介绍 GCD 和 Operations——iOS 开发中最主要的处理并发编程的两套 API。除了理论，在面试实战题部分还会介绍如何将并发编程运用在实际 App 的开发中。最后，还会介绍如何 DeBug 并发编程问题：包括如何在 Xcode 中观察线程信息，以及如何发现并解决并发编程问题。

在 iOS 开发中，对于并发操作有哪 3 种方式

关键词：#NSThread　#GCD　#Operations

在 iOS 开发中，基本有以下 3 种方式可实现并发操作。

- **NSThread**：可以最大限度地掌控每一个线程的生命周期。但是，也需要开发者手动管理所有的线程活动，比如创建、同步、暂停、取消等，其中手动加锁操作的挑战性很大。NSThread 总体使用场景很小，基本是在开发底层的开源软件或是测试时使用。

- **GCD（Grand Central Dispatch）**：苹果公司推荐的方式，它将线程管理推给了系统，用的是名为 Dispatch Queue 的队列；开发者只要定义每个线程需要执行的工作即可；所有的工作都是先进先出，每一个 block 运转速度极快（在纳秒级别）。使用 GCD 主要是为了追求高效处理大量并发数据，如图片异步加载、网络请求等。

- **Operations**：与 GCD 类似。虽然是 Operation Queue 队列实现，但是它并不局限于先进先出的队列操作。Operations 提供了多个接口可以实现暂停、继续、终止、优先顺序、依赖等复杂操作，比 GCD 更加灵活。Operations 的应用场景最广，在效率上每个 Operation 处理速度较快（毫秒级别），几乎所有的基本线程操作都可以实现。

比较关键词：Serial，Concurrent，Sync 和 Async

关键词：#多任务　#阻塞线程

对于 Serial, Concurrent, Sync 和 Async 这 4 个关键词，前两个关键词（Serial/Concurrent）构成一对，后两个关键词（Sync/Async）构成一对，下面分别介绍。

- **Serial/Concurrent**：声明队列的属性是串行的还是并行的。串行队列（Serial Queue）指在同一时间内，队列中只能执行一个任务，当前任务执行完后才能执行下一个任务。在串行队列中只有一个线程。并行队列（Concurrent Queue）允许多个任务在同一个时间同时进行，在并行队列中有多个线程。串行队列的任务一定是按开始的顺序结束，而并行队列的任务并不一定会按照开始的顺序结束。

- **Sync/Async**：声明任务是同步执行的还是异步执行的。同步（Sync）会把当前的任务加到队列中，等到任务执行完成，线程才会返回继续运行。也就是说，同步会阻塞线程。异步（Async）也会把当前的任务加到队列中，但它会立刻返回，无须等任务执行完成，也就是说异步不会阻塞线程。

无论是串行队列还是并行队列，都可以执行同步或异步操作。注意，在串行队列中执行同步操作容易造成死锁，在并行队列中则不用担心这个问题。异步操作无论是在串行队列中执行还是在并行队列中执行，都可能出现竞态条件的问题；同时，异步操作经常与逃逸闭包一起出现在 API 的设计中。

串行队列的代码实战

关键词：#串行　#同步　#异步

以下代码均在串行队列中发生，执行之后会打印出什么？

```
// 串行同步
serialQueue.sync {
  print(1)
}
print(2)
serialQueue.sync {
  print(3)
}
print(4)
```

```
// 串行异步
serialQueue.async {
  print(1)
}
print(2)
serialQueue.async {
  print(3)
}
print(4)

// 串行异步中嵌套同步
print(1)
serialQueue.async {
  print(2)
  serialQueue.sync {
    print(3)
  }
  print(4)
}
print(5)

// 串行同步中嵌套异步
print(1)
serialQueue.sync {
  print(2)
  serialQueue.async {
    print(3)
  }
  print(4)
}
print(5)
```

首先，在串行队列上进行同步操作时，所有任务将顺序发生，所以，第一段代码的打印结果一定是 1234。

其次，在串行队列上进行异步操作，此时任务完成的顺序并不保证。所以，可能会打印出几种结果：1234，2134，1243，2413，2143。注意，1 一定会在 3 之前被打印出来，因为 1 在 3 之前被派发，串行队列一次只能执行一个任务，所以一旦派发完成就执行任务。同

理，2 一定会在 4 之前被打印出来，以及 2 一定会在 3 之前被打印出来。

接着，在同一个串行队列中进行异步、同步嵌套，这里会构成死锁，所以只会打印出 12。

最后，在串行队列中进行同步、异步嵌套，不会构成死锁。此时会打印出两个结果：12345，12435。注意，同步操作保证了 3 一定会在 4 之前被打印出来。

并行队列的代码实战

关键词：#并行　#同步　#异步

以下代码均在并行队列中发生，执行之后会打印出什么？

```
// 并行同步
concurrentQueue.sync {
  print(1)
}
print(2)
concurrentQueue.sync {
  print(3)
}
print(4)

// 并行异步
concurrentQueue.async {
  print(1)
}
print(2)
concurrentQueue.async {
  print(3)
}
print(4)

// 并行异步中嵌套同步
print(1)
concurrentQueue.async {
  print(2)
  concurrentQueue.sync {
    print(3)
```

```
  }
  print(4)
}
print(5)


// 并行同步中嵌套异步
print(1)
concurrentQueue.sync {
  print(2)
  concurrentQueue.async {
    print(3)
  }
  print(4)
}
print(5)
```

首先，在并行队列中进行同步操作，所有任务将顺序执行、顺序完成，所以，第一段代码的打印结果一定是 1234。

其次，在并行队列中进行异步操作时，其效果与之前串行列队类似。所以，可能会打印出来这几种结果：1234 ，2134，1243，2413，2143。

接着，在同一个并行队列中进行异步、同步嵌套。这里不会构成死锁，因为同步操作只会阻塞一个线程，而并行队列对应多个线程。这里会打印出 4 个结果：12345，12534，12354，15234。注意，同步操作保证了 3 一定会在 4 之前被打印出来。

最后，在并行队列中进行同步、异步嵌套，不会构成死锁。而且由于是并行队列，所以，在运行异步操作时也同时会运行其他操作。这里会打印出 3 个结果：12345，12435，12453。这里的同步操作保证了 2 和 4 一定会在 3 之前被打印出来。

举例说明 iOS 并发编程中的三大问题

关键词：#竞态条件　#优先倒置　#死锁问题

在并发编程中，一般会面对三个问题：竞态条件、优先倒置和死锁问题。针对 iOS 开发，

它们的具体定义为：

- **竞态条件（Race Condition）**：指两个或两个以上线程对共享的数据进行读写操作时，最终的数据结果不确定的情况。例如以下代码：

```
var num = 0
DispatchQueue.global().async {
  for _ in 1…10000 {
    num += 1
  }
}

for _ in 1…10000 {
  num += 1
}
```

最后的计算结果 num 很有可能小于 20000，因为其操作为非原子操作。在上述两个线程对 num 进行读写时，其值会随着进程执行顺序的不同而产生不同的结果。

- **优先倒置（Priority Inverstion）**：指低优先级的任务会因为各种原因先于高优先级的任务执行。例如以下代码：

```
var highPriorityQueue = DispatchQueue.global(qos: .userInitiated)
var lowPriorityQueue = DispatchQueue.global(qos: .utility)

let semaphore = DispatchSemaphore(value: 1)

lowPriorityQueue.async {
  semaphore.wait()
  for i in 0...10 {
    print(i)
  }
  semaphore.signal()
}

highPriorityQueue.async {
  semaphore.wait()
  for i in 11...20 {
    print(i)
```

```
  }
  semaphore.signal()
}
```

上述代码如果没有 semaphore，则高优先级的 highPriorityQueue 会优先执行，所以程序会优先打印完 11~20。而加了 semaphore 之后，低优先级的 lowPriorityQueue 会先挂起 semaphore，高优先级的 highPriorityQueue 就只有等 semaphore 被释放才能再执行打印操作。

也就是说，低优先级的线程可以锁上某种高优先级线程需要的资源，从而迫使高优先级的线程等待低优先级的线程。

- **死锁问题（Dead Lock）**：指两个或两个以上的线程，它们之间互相等待彼此停止执行，以获得某种资源，但是没有一方会提前退出的情况。在 iOS 开发中，有一个经典的例子就是两个 Operation 互相依赖：

```
let operationA = Operation()
let operationB = Operation()

operationA.addDependency(operationB)
operationB.addDependency(operationA)
```

还有一种经典的例子：在对同一个串行队列中进行异步、同步嵌套时：

```
serialQueue.async {
  serialQueue.sync {
  }
}
```

因为串行队列一次只能执行一个任务，所以，首先它会把异步 block 中的任务派发执行，当进入 block 中时，同步操作意味着阻塞当前队列 。而此时外部 block 正在等待内部 block 操作完成，而内部 block 又阻塞其操作完成，即内部 block 在等待外部 block 操作完成。所以，串行队列在等待自己释放资源，构成死锁。这也提醒了我们，千万不要在主线程中用同步操作。

竞态条件的代码实战

关键词：#竞态条件　#thread sanitizer

以下代码有什么隐患？

```
func getUser(id: String) throws -> User {
  return try storage.getUser(id)
}

func setUser(_ user: User) throws {
  try storage.setUser(user)
}
```

上面这段代码的功能是读写用户信息。虽然这段代码乍一看没有什么问题，但是一旦多线程涉及读写操作，就会产生竞态条件（Race Condition）。解决的方法是打开 Xcode 中的线程检测工具 Thread Sanitizer（在 Xcode 的 scheme 中勾选“Thread Sanitizer”选项即可），它会检测出代码中出现竞态条件之处，并提醒我们修改。

对于读写问题，一般有以下 3 种处理方式。

第 1 种是用串行队列，无论是读操作还是写操作，同一时间只能进行一个操作，这样就保证了队列的安全。但是缺点是速度慢，尤其是在有大量读、写操作时，每次只能进行单个读操作或写操作，效率实在太低了。修改代码如下：

```
func getUser(id: String) throws -> User {
  return serialQueue.sync {
    return try storage.getUser(id)
  }
}
func setUser(_ user: User) throws {
  try serialQueue.sync {
    try storage.setUser(user)
  }
}
```

第 2 种是用并行队列配合异步操作完成。异步操作由于会直接返回结果，所以必须配合逃

逸闭包来保证后续操作的合法性。

```
enum Result<T> {
  case value(T)
  case error(Error)
}
```

```
func getUser(id: String, completion: (Result<User>) - Void) {
  try serialQueue.async {
    do {
      user = try storage.getUser(id)
      completion(.value(user))
    } catch {
      completion(.error(error))
    }
  }

  return user
}
```

```
func setUser(_ user: User, completion: (Result<()>) -> Void) {
  try serialQueue.async {
    do {
      try storage.setUser(user)
      completion(.value(()))
    } catch {
      completion(.error(error))
    }
  }
}
```

第 3 种是用并行队列，在进行读操作时，用 sync 直接返回结果；在进行写操作时，用 barrier flag 来保证此时并行队列只进行当前的写操作（类似于将并行队列暂时转为串行队列），而无视其他操作。示例代码如下：

```
enum Result<T> {
  case value(T)
  case error(Error)
}
```

```
func getUser(id: String) throws -> User {
  var user: User!
  try concurrentQueue.sync {
    user = try storage.getUser(id)
  }

  return user
}

func setUser(_ user: User, completion: (Result<()>) -> Void) {
  try concurrentQueue.async(flags: .barrier) {
    do {
      try storage.setUser(user)
      completion(.value(()))
    } catch {
      completion(.error(error))
    }
  }
}
```

试比较 GCD 中的方法：dispatch_async，dispatch_after，dispatch_once 和 dispatch_group

关键词：#异步　#延时　#单例　#线程同步

首先要明确 dispatch_async, dispatch_after, dispatch_once 和 dispatch_group 的用法。

这几个关键词都是在 Objective-C 编程中出现的。它们分别有如下用法：

- dispatch_async：用于对某个线程进行异步操作。异步操作可以让我们在不阻塞线程的情况下，充分利用不同线程和队列来处理任务。例如，当需要从网络端下载图片，然后将图片赋予某个 UIImageView 时，就可以用到 dispatch_async:

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
  UIImage *image = [client fetchImageFromURL: url];
  dispatch_async(dispatch_get_main_queue(), ^{
    self.imageView.image = image;
```

```
  });
});
```

- dispatch_after：一般用于主线程的延时操作。例如，要将一个页面的导航标题——“等待”在两秒后改为“完成”，则可以用 dispatch_after 来实现：

```
self.title = @"等待";
double delayInSeconds = 2.0;
dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(delayInSeconds
* NSEC_PER_SEC));
dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
  self.title = @"完成";
});
```

- dispatch_once：用于确保单例的线程安全。它表示修饰的区域只会访问一次，这样在多线程情况下，类也只会初始化一次，确保了 Objective-C 中单例的原子化。

```
+ (instancetype)sharedManager {
  static Manager *sharedManager = nil;
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{
    sharedManager = [[Manager alloc] init];
   });
   return sharedManager;
}
```

- dispatch_group：一般用于多个任务同步。一般用法是当多个任务关联到同一个群组（group）后，所有的任务在执行完后，再执行一个统一的后续工作。注意，dispatch_group_wait 是一个同步操作，它会阻塞线程。

```
dispatch_group_t group = dispatch_group_create();

dispatch_group_async(group,
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
  NSLog(@"开始做任务 1！");
});

dispatch_group_async(group,
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
```

```
  NSLog(@"开始做任务 2! ")
});

dispatch_group_notify(group, dispatch_get_main_queue(), ^{
  NSLog(@"任务 1 和任务 2 都完成了! ")
});
```

GCD 中全局（global）队列有哪几种优先级

关键词：#QOS

首先，全局队列肯定是并发队列。如果不指定优先级，则为默认（default）优先级。另外，优先级还有 background，utility，user-Initiated，unspecified 和 user-Interactive。下面按照优先级顺序从低到高进行排列。

- **background**：用来处理特别耗时的后台操作，例如同步、备份数据。
- **utility**：用来处理需要一些时间而又不需要立刻返回结果的操作。其特别适用于异步操作，例如下载、导入数据。
- **default**：默认优先级。一般来说开发者应该指定优先级。Default 属于特殊情况。
- **user-Initiated**：用来处理用户触发的需要立刻返回结果的操作，比如，打开用户点击的文件。
- **user-Interactive**：用来处理与用户交互的操作。其一般用于主线程，如果不及时响应，则有可能阻塞主线程的操作。
- **unspecified**：未确定优先级，由系统根据不同环境推断。比如，使用过时的 API 不支持优先级时，就可以设定为未确定优先级。unspecified 属于特殊情况。

试比较 Operations 中的关键词：Operation，BlockOperation 和 OperationQueue

关键词：#Operation

Operations 是 iOS 中除 GCD 外又一种实现并发编程的方式。它将单个任务算作一个 Operation，然后放在 OperationQueue 队列中进行管理和运行。其中最常用的 3 个关键词就是 Operation，BlockOperation 和 OperationQueue。

- **Operation**：指代一系列工作或者任务。Operation 本身是一个抽象类，一般通过集成它来完成自定义的工作。每个单独的 Operation 有 4 种状态：准备就绪（Ready）、执行中（Executing）、暂停（Cancelled）和完成（Finished）。下面就是一个自定义的图片转换的 Operation 子类：

```
class ImageFilterOperation: Operation {
  var inputImage: UIImage?
  var outputImage: UIImage?

  override func main() {
    outputImage = filter(image: inputImage)
  }
}
```

- **BlockOperation**：系统定义的一个 Operation 的子类。它本身在默认权限的全局队列上运行，负责并行执行多个任务。其中任务之间互不依赖，同时，BlockOperation 也可以像 dispatch_group 一样同步管理多个任务。例如下面的示例代码：

```
let multiTasks = BlockOperation()

multiPrinter.completionBlock = {
  print("所有任务完成!")
}

multiTasks.addExecutionBlock {  print("任务 1 开始"); sleep(1) }
multiTasks.addExecutionBlock {  print("任务 2 开始"); sleep(2) }
multiTasks.addExecutionBlock {  print("任务 3 开始"); sleep(3) }

multiPrinter.start()
```

- **OperationQueue**：负责安排和运行多个 Operation 的队列。但是它并不局限于先进先出的队列操作。它提供了多个接口可以实现暂停、继续、中止、优先顺序、依赖等复杂操作。同时，还可以通过设置 maxConcurrentOperationCount 属性来区分其是串行的

还是并行的，例如下面的示例代码：

```
Let multiTaskQueue = OperationQueue()

multiTaskQueue.addOperation { print("任务 1 开始"); sleep(1) }
multiTaskQueue.addOperation { print("任务 2 开始"); sleep(2) }
multiTaskQueue.addOperation { print("任务 3 开始"); sleep(3) }

multiTaskQueue.waitUntilAllOperationsAreFinished()
```

如何在 OperationQueue 中取消某个 Operation

关键词：#cancel()　#isCancelled

在 Operation 抽象类中，有一个 cancel()方法，它做的唯一一个工作就是将 Operation 的 isCancelled 属性从 false 改为 true。由于它并不会真正深入代码将某个具体执行的工作暂停，所以必须要利用 isCancelled 属性的变化来暂停 main()方法中的工作。

举一个例子，这里有一个求和整型数组的操作，其对应的 Operation 如下：

```
class ArraySumOperation: Operation {
  let nums: [Int]
  var sum: Int

  init(nums: [Int]) {
    self.nums = nums
    self.sum = 0
    super.init()
  }

  override func main() {
    for num in nums {
      sum += num
    }
  }
}
```

如果要运行该 Operation，则应该将其加入 OperationQueue 中：

```
let queue = OperationQueue()
let sumOperation = ArraySumOperation(nums: Array(1…1000))

// 一旦加入 OperationQueue 中，Operation 就开始执行
queue.addOperation(sumOperation)

// 打印出 500500（1+2+3+…+1000）
sumOperation.completionBlock = { print(sumOperation.sum) }
```

如果要取消，则应该调用 cancel()方法，但是它只会将 isCancelled 改成 false，而我们要利用 isCancelled 的状态控制 main()中的操作，所以可以将 ArraySumOperation 改成如下形式：

```
class ArraySumOperation: Operation {
  let nums: [Int]
  var sum: Int

  init(nums: [Int]) {
    self.nums = nums
    self.sum = 0
    super.init()
  }

  override func main() {
    for num in nums {
      if isCancelled {
        return
      }
      sum += num
    }
  }
}
```

此时，运行 Operation，就会得到不同结果：

```
let queue = OperationQueue()
let sumOperation = ArraySumOperation(nums: Array(1...1000))

// 一旦加入 OperationQueue 中，Operation 就开始执行
queue.addOperation(sumOperation)
```

```
sumOperation.cancel()

// sumOperation 在彻底完成前已经暂停，sum 值小于 500500
sumOperation.completionBlock = { print(sumOperation.sum) }
```

同时，OperationQueue 还有 cancelAllOperations()方法可以调用，它相当于是对所有在该队列上工作的 Operation，都调用其 cancel()方法。

在实际开发中，主线程和其他线程有哪些使用场景

关键词：#UI　#耗时

主线程一般用于负责 UI 相关操作，如绘制图层、布局等。很多 UIKit 相关的控件如果不在主线程中操作，则会产生未知效果。Xcode 中的 Main Thread Checker 可以将相关问题检测出来并报错。

其他线程，例如后天线程，一般用来处理比较耗时的工作。数据解析、复杂计算、图片的编码和解码等都属于耗时的工作，应该放在其他线程中处理。如果放在主线程中，则由于是串行队列，会直接阻塞主线程的 UI 操作，影响用户体验。

4.5　设计模式

很多刚入门的 iOS 开发者经过短期训练，都可以熟练地调用各种 API，这时候，写一个 tableView，实现一个小动画，独立完成一个交互的功能，已经不在话下。但同时，iOS 开发者也会遇到技术上的第一个瓶颈——即拥有独立开发一个功能的水平，却似乎并未达到独立开发一个 App 的水平；看似什么都会做，什么都能做，却总是不能在第一时间想到最佳方案；功能是完成了，然而效率不是很高，代码逻辑在日后也可能需要返工重构。

我认为，突破这个瓶颈的捷径就是掌握设计模式。设计模式是前人总结的、面对开发中常见问题的解决方案——它们行之有效，便于理解，适合举一反三。简单点儿说，设计模式就是程序开发的套路和模板。熟练掌握设计模式，可以提高开发效率，节省开发时间。这样，我们就可以站在前人的肩膀上，去研究、解决那些具有挑战性和未曾解决过的问题。