答疑注意点

1、圈复杂度=独立路径数目（从左到右依次加层）

2、类图关系：关联（写关联属性、关联类）、聚合、继承——要写多重性

3、时序图（LOOP）实例化对象，返回可以不写标识

4、部署图（一定考）：

5、架构设计：分层、异步、缓存、并发、负载均衡——体系图！！！

用例优先级问题！！！！！

实例化谁就指向谁：记录的东西

后台服务器一般算本身系统（要自己写代码）、也可能是外部（银行直接调用）

消息队列：更好的伸缩性，削峰填谷，失败隔离与自我修复，解耦

题型分布

1、选择题：10（10选择、慕课选择）

2、简答：架构风格！！、过程开发模型、git、需求是否合理

3、大题：用例图、领域类图、时序图、部署图、黑盒测试、白盒测试（只用写路径）

先把选择题过完/13

用例图->

1-JAVA

JAVA数据定义

1. 基本数据类型（Primitive Types）

Java中有八种基本数据类型，它们是最简单、最直接的数据类型，不属于对象类型。这些类型有固定的大小和预定义的操作。基本数据类型包括：

- **byte**：8位整数类型
- **short**：16位整数类型
- **int**：32位整数类型
- **long**：64位整数类型
- **float**：32位单精度浮点数
- **double**：64位双精度浮点数
- **char**：16位Unicode字符
- **boolean**：表示布尔值（true或false）

2. 引用类型（Reference Types）

引用类型包括类、接口、数组等。它们是通过引用（即指针）来访问实际对象的。

2.1 字符串类（String）

`String`类是Java中用于表示字符序列的类，是一种特殊的引用类型。它有一些特殊的属性和方法，用于处理字符串。

- 特点：
  - 不可变性（Immutable）：一旦创建，`String`对象的值不能改变。
  - 常量池（String Pool）：为了优化内存使用，JVM会将字符串字面量存储在一个常量池中。
- 常用方法：
  - `length()`：返回字符串的长度。
  - `charAt(int index)`：返回指定位置的字符。
  - `substring(int beginIndex, int endIndex)`：返回字符串的子字符串。
  - `equals(Object obj)`：比较两个字符串的内容是否相同。
  - `toUpperCase()` / `toLowerCase()`：将字符串转换为大写或小写。
  - `trim()`：去掉字符串两端的空白字符。
  - `replace(CharSequence target, CharSequence replacement)`：替换字符串中的字符序列。

2.2 基本类型的包装类（Wrapper Classes）

每个基本数据类型都有对应的包装类，它们都是引用类型，这些类位于`java.lang`包中。包装类主要用于将基本类型的数据包装成对象，以便在需要对象的场合使用。

- **byte** → `Byte`
- **short** → `Short`
- **int** → `Integer`
- **long** → `Long`
- **float** → `Float`
- **double** → `Double`
- **char** → `Character`
- **boolean** → `Boolean`

示例：

```
int primitiveInt = 10;
Integer wrapperInt = Integer.valueOf(primitiveInt); // 使用包装类
int unboxedInt = wrapperInt.intValue(); // 将包装类转换回基本类型
```

3. 集合框架（Collection Framework）

集合框架用于存储和操作一组对象（元素）。它包含多个接口和类，用于不同的数据结构和操作需求。

3.1 集合接口和实现类

- **List 接口**：有序集合，允许重复元素。
  - 实现类：`ArrayList`, `LinkedList`, `Vector`, `Stack`
- **Set 接口**：无序集合，不允许重复元素。
  - 实现类：`HashSet`, `LinkedHashSet`, `TreeSet`
- **Queue 接口**：通常用于存储按顺序处理的元素。
  - 实现类：`LinkedList`, `PriorityQueue`, `ArrayDeque`
- **Deque 接口**：双端队列，可以从两端添加或移除元素。
  - 实现类：`LinkedList`, `ArrayDeque`
- **Map 接口**：键值对存储，每个键唯一。
  - 实现类：`HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`, `ConcurrentHashMap`

4. 其他常用类

- **StringBuilder / StringBuffer**：用于创建可变字符串。`StringBuilder`是非线程安全的，而`StringBuffer`是线程安全的。
- **Arrays**：包含用于操作数组的各种方法，如排序和搜索。

JAVA语法

1. 基本语法

1.1 数据类型

- **基本数据类型**：
  - 整数类型：`byte`, `short`, `int`, `long`
  - 浮点类型：`float`, `double`
  - 字符类型：`char`
  - 布尔类型：`boolean`
- **引用类型**：
  - 类：`String`, `Integer`, `Double`, 等
  - 接口
  - 数组：`int[]`, `String[]`

1.2 变量声明

```
int number = 10;
String text = "Hello, World!";
```

1.3 控制结构

- **条件语句**:

```
if (condition) {
    // ...
} else if (anotherCondition) {
    // ...
} else {
    // ...
}
```

- **循环语句**:

```
// for loop
for (int i = 0; i < 10; i++) {
    // ...
}

// while loop
while (condition) {
    // ...
}

// do-while loop
do {
    // ...
} while (condition);
```

2. 面向对象编程

2.1 类和对象

- **类声明**:

```
public class MyClass {
    // 属性（成员变量）
    private int number;
    private String text;

    // 构造函数
    public MyClass(int number, String text) {
        this.number = number;
        this.text = text;
    }

    // 方法
    public void display() {
        System.out.println(text + ": " + number);
    }
}
```

- **创建对象**:

```
MyClass obj = new MyClass(10, "Hello");
obj.display();
```

2.2 继承

- 父类和子类：

```
public class ParentClass {
    public void display() {
        System.out.println("Parent class method");
    }
}

public class ChildClass extends ParentClass {
    @Override
    public void display() {
        System.out.println("Child class method");
    }
}

ChildClass child = new ChildClass();
child.display(); // 输出: Child class method
```

2.3 接口

- 接口声明和实现：

```
public interface MyInterface {
    void method1();
    void method2();
}

public class MyClass implements MyInterface {
    @Override
    public void method1() {
        // 实现方法1
    }

    @Override
    public void method2() {
        // 实现方法2
    }
}
```

3. 集合框架

3.1 常用集合

- **List**：

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
```

- **Set**:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
```

- **Map**:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
```

4. 异常处理

- try-catch 语句:

```
try {
    // 可能抛出异常的代码
} catch (ExceptionType e) {
    // 处理异常
} finally {
    // 无论是否发生异常都会执行的代码
}
```

5. 文件输入输出

- **读取文件**:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **写入文件**:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("file.txt"))) {
            bw.write("Hello, World!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

6. 多线程

- **创建线程**:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}

MyThread thread = new MyThread();
thread.start();
```

- **实现Runnable接口**:

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable is running");
    }
}

Thread thread = new Thread(new MyRunnable());
thread.start();
```

- **注解：** `@Override`
- **翻译：** 覆盖（重写）
- **作用：** 标明子类中的方法是对父类方法的覆盖或实现接口的方法

2-架构

1.缓存和分布式集群的差别

缓存（Cache）

原理：

- 缓存通过存储数据的临时副本来减少对后端系统的直接请求。这些数据通常是计算成本高或者请求频率高的数据，例如数据库查询结果或者重复的API调用输出。
- 当一个请求到达系统时，缓存层首先检查所请求的数据是否已存储在缓存中。如果是，数据可以直接从缓存返回，避免了对数据库或后端服务的昂贵调用。
- 有效的缓存策略可以显著减少数据检索时间，从而加快响应速度。

优点：

- 显著减少对后端资源的请求，降低了系统负载。
- 减少数据处理和传输时间，直接从内存中获取数据比从数据库读取快得多。

缺点：

- 缓存数据需要定期更新，以避免数据过时。
- 管理缓存一致性和数据同步可能会增加系统的复杂度。

分布式集群

原理：

- 分布式集群通过增加计算节点来分散负载，提升系统的处理能力。这意味着当多个请求同时到达时，这些请求可以被分配到不同的服务器上并行处理。
- 通过负载均衡器，分布式集群能够有效地管理请求流向，确保没有单一节点的过载，从而优化资源的使用和提高响应速度。
- 分布式系统还可以通过地理分散部署来降低网络延迟，使用户能够接近数据源或处理节点，进一步改善响应时间。

优点：

- 提升系统的整体处理能力和可扩展性。
- 能够通过增加节点简单地扩展系统，适应不断增长的用户和数据量。

缺点：

- 增加了系统的维护和管理复杂度。
- 可能涉及更多的网络通信和数据一致性问题。

差异比较

- **目标差异**：缓存主要通过减少请求的处理时间来提高响应速度，而分布式集群通过增加处理能力来应对大量并发请求。
- **资源利用**：缓存使得数据可以快速访问，主要优化的是访问延迟；分布式集群通过多节点处理，主要提升的是计算能力和负载分散。

- **扩展性**：缓存的扩展性主要受限于缓存策略和内存大小，而分布式集群的扩展性更高，可以通过添加更多的服务器来提升。
- **成本和复杂度**：通常来说，维护一个有效的缓存系统可能比构建一个分布式集群更简单，成本也更低。然而，分布式集群提供了更强的可扩展性和故障容错性。

2.面向NFR的架构设计：2014期末

1）非功能指标的折中

在面向非功能需求（NFRs）的软件架构设计中，部署策略的选择对软件的性能、可用性、可维护性、成本效益等方面有显著影响。对于提到的三种音乐库部署策略，我们可以从多个非功能指标进行分析：

策略1: 音乐库完全部署于App端

优势：

1. **离线访问**：用户无需互联网连接即可访问音乐库，提高了使用便捷性。
2. **响应时间**：本地存储的内容可以即时访问，无需等待网络响应，提高了用户体验。

劣势：

1. **存储需求**：音乐库可能非常庞大，对移动设备的存储空间要求很高，可能不现实。
2. **数据更新和同步**：更新音乐库需要完全重新下载，这可能非常耗费流量和时间，且难以保持最新状态。
3. **成本**：在每个设备上维护完整的音乐库增加了数据分发和存储成本。

策略2: 音乐库完全部署于服务器端

优势：

1. **易于维护和更新**：所有更新只需在服务器端进行，易于管理且对用户透明。
2. **节省设备存储**：不占用用户设备的存储空间。
3. **可扩展性**：服务器端更易于扩展，可以根据需求增加更多服务器和存储资源。

劣势：

1. **依赖网络**：用户必须持续连接到互联网才能访问音乐，可能影响在网络不稳定的环境中的使用。
2. **延迟**：可能会因网络延迟影响播放体验，特别是在网络质量差的地区。

策略3: 服务器端部署完整音乐库，App端部署用户感兴趣的子库

优势：

1. **灵活性**：结合了前两种策略的优点，用户可以在线访问完整库，同时离线访问已下载的音乐。
2. **节省带宽和成本**：只下载用户感兴趣的音乐，减少了数据传输量和成本。
3. **适应性**：可以根据用户的喜好和行为模式动态调整存储在本地的内容。

劣势：

1. **复杂性**：需要有效管理本地和服务器端的数据同步，增加了系统的复杂性。
2. **资源使用**：虽然比完全本地存储节省，但仍然需要占用一定的本地存储空间。
3. **实现难度**：需要更智能的客户端应用来管理数据缓存和更新，增加了开发和维护的难度。

结论

每种策略都有其适用的场景和权衡点。在选择部署策略时，应考虑目标用户群的具体需求、设备能力、网络状况以及成本效益。策略3通常提供最大的灵活性和用户体验，但也要求更高的技术实现标准和维护成本。

2. 数据一致性与性能优化策略

增量同步：

- 通过仅同步上次更新后发生变化的数据，减少每次同步的数据量，同时确保数据的实时性。
- 可以利用百度音乐库提供的API检查数据的最后更新时间戳，确定需要同步的数据。

配置同步频率：

- 允许用户或管理员根据需要设置同步频率，如在夜间或流量低峰时段进行同步，减少对应用性能的影响。
- 在网络状况较差或系统负载高时自动调整同步频率，以保护应用性能。

数据量控制：

- 提供接口设置每次同步的数据量上限，避免在一次同步中传输大量数据，影响应用响应速度和用户体验。
- 根据网络状况和设备性能动态调整数据量。

智能预加载与缓存：

- 对于高频访问的音乐数据实行预加载和缓存策略，减少对实时网络请求的依赖。
- 缓存机制可以结合本地存储管理，定期清理不常用的缓存数据，优化存储使用。

3. 实现技术

- **RESTful API**：使用RESTful接口与百度音乐库通信，利用HTTP的GET方法来拉取更新数据。
- **JSON格式**：同步数据使用JSON格式，便于处理和解析。
- **后台线程**：同步操作在后台线程进行，避免阻塞主线程，影响用户界面的响应。
- **差异比对算法**：采用高效的数据比对算法，确保快速准确地识别出需要更新的数据。

3)架构设计决策

1. 动态资源扩展和负载均衡

设计决策：

- 实施自动扩展的云基础设施，例如使用AWS Auto Scaling、Google Compute Engine的自动扩展器或Azure Virtual Machine Scale Sets。这些服务可以根据负载增加或减少实例的数量。
- 部署负载均衡器（如AWS Elastic Load Balancer或Nginx），以均匀分配到达的请求到各个服务器实例。

依据：

- **自动扩展** 确保在访问量激增时，系统可以动态增加更多的服务器资源来处理增加的请求，而在流量低时自动缩减资源，优化成本效益。
- **负载均衡** 通过合理分配网络流量和请求，避免任何单一服务器因超负荷而导致性能瓶颈或服务中断，从而提高整体的系统可用性和响应速度。

2. 数据缓存策略与内容分发网络（CDN）

设计决策：

- 实施分层的缓存策略，包括内存缓存（如Redis）和应用级缓存，以减少对后端数据库的直接查询，快速响应用户请求。
- 使用内容分发网络（CDN），如Akamai或Cloudflare，以将音乐数据和静态内容（如应用资源、图片等）缓存于离用户较近的地理位置。

依据：

- **缓存** 可显著减少数据检索时间，对于频繁访问的数据（如热门歌曲或用户偏好列表），使用缓存可以直接从内存中快速提供数据，减轻后端服务器的负载。
- **CDN** 通过将内容存储在全球多个位置，确保即便在用户访问高峰，内容也能从最近的节点快速加载，降低了对原始服务器的直接请求，加速了数据的加载速度，同时提高了用户体验和应用的扩展性。

3-需求

1.怎么向用户提问

需求[9]：自动计算关键日期并提前发出提醒，允许上传更多照片

向用户提出的问题：

1. **关键日期定义**："您能否具体说明哪些日期应被视为'关键日期'？例如，是否包括生日、特定的成长里程碑日等？"
2. **上传照片的数量**："在这些关键日期，您希望能上传多少张照片？是否有一个具体的上限或范围？"

假设用户回答：

- "关键日期包括生日、首次走路、首次说话等成长里程碑。在这些日期，我希望能上传至少10张照片，最多不超过20张。"

需求修改：

- **改进的需求**："系统将自动识别并标记出如孩子的每年生日、首次走路、首次说话等成长里程碑作为关键日期，并在这些日期的前一天通过应用内通知提醒用户。在这些关键日期中，用户可以上传最少10张、最多20张照片。"

需求[10]：按关键字、日期、日期段检索照片

向用户提出的问题：

1. **关键字范围**："您希望通过哪些关键字来检索照片？是否包括描述、事件名称或其他任何特定的标签？"
2. **检索功能的操作界面**："您期望如何操作这些检索功能？是否希望通过一个搜索框输入关键字，还是通过选择日期来过滤照片？"

假设用户回答：

- "我希望可以通过照片的描述文字、具体事件名称、以及日期来检索。我想要一个搜索框来输入关键字，并且可以选择具体的日期或日期范围来过滤照片。"

需求修改：

- **改进的需求**："用户可以通过一个搜索框输入照片的描述文字或事件名称来检索照片。用户也可以选择具体的日期或输入日期范围来过滤搜索结果。搜索结果将按照日期顺序展示，可视化在时间线或日历视图中。"

4-测试

一、黑盒测试的扩展性（无法直接测试的情况下的解决办法）：2014期末五

(1) 需从测试执行过程中采集的数据

为了有效判断推荐算法的优劣，除了关注算法直接输出的“一组推荐的音乐”之外，还应该收集以下数据：

1. **用户反馈和互动数据**：
   - **直接反馈**：用户对推荐歌曲的喜欢或不喜欢的点击次数。
   - **间接互动**：包括歌曲的播放次数、完整播放率、跳过率、保存或添加到播放列表的频率。
2. **用户行为日志**：
   - 记录用户与推荐系统互动的详细日志，包括搜索行为、浏览历史以及对推荐列表的响应。
3. **系统性能指标**：
   - 响应时间：系统生成推荐列表所需的时间。
   - 系统负载：在高并发时段系统的处理能力和稳定性。
4. **推荐多样性和新颖性指标**：
   - 多样性：推荐结果中不同类型或风格歌曲的比例。
   - 新颖性：推荐给用户他们未曾听过的歌曲的比例。

(2) 对算法进行优劣评判的标准

基于上述收集的数据，可以根据以下标准评估推荐算法的性能：

1. **用户满意度**：
   - 高用户点击“喜欢”比率和正反馈指标表明推荐系统能有效匹配用户喜好。
2. **用户参与度**：
   - 高的播放次数和完整播放率表明推荐内容具有吸引力。
3. **系统效率**：
   - 快速的响应时间和在高流量期间的稳定表现。
4. **内容多样性和新颖性**：
   - 一定比例的新颖歌曲和风格多样的推荐列表通常意味着更高的用户探索性和满意度。

(3) 如何评判所设计的一组测试用例是充分的

一个充分的测试用例设计应满足以下条件：

1. **全面性**：
   - 包括不同用户类型（个人听众、DJ、唱片公司）的场景。
   - 覆盖不同时间段，包括高峰和低峰时段。
   - 测试用例应覆盖算法对新老用户的推荐效果。
2. **实际场景的模拟**：
   - 测试用例应模拟真实世界中的用户行为，如不同的历史听歌记录、不同的兴趣偏好等。
3. **反馈机制的测试**：
   - 验证算法是否能根据用户的“喜欢”或“不喜欢”反馈进行适当的调整。
4. **性能测试**：
   - 在系统资源限制下（如服务器负载高）测试推荐算法的响应时间和准确性。

5-OO设计与分析

一、2016年期末：伪代码

1.特定日历上的查重

该操作用于检测在给定日历中指定日期的活动是否存在时间上的重叠。

```
函数 findOverlaps 在 Calendar 类中:
输入: date: Date
输出: List<Tuple<Event, Event>>

开始
    创建空列表 overlaps
    对于 calendar.events 中的每个事件 event1
        对于 calendar.events 中的每个事件 event2
            如果 event1 不等于 event2 并且 event1.startTime < event2.endTime 且
event1.endTime > event2.startTime
                将 (event1, event2) 添加到 overlaps
    返回 overlaps
结束
```

2. 跨日历的查重

该操作检查用户所有日历中的活动在给定日期是否存在时间上的重叠。

```
函数 crossCalendarOverlapCheck 在 User 类中:
输入: date: Date
输出: List<Tuple<Event, Event>>

开始
    创建空列表 allOverlaps
    对于 user.calendars 中的每个 calendar1
        对于 user.calendars 中的每个 calendar2
            如果 calendar1 等于 calendar2
```

```
                继续到下一次循环
            获取 calendar1 的重叠情况 overlaps1 = calendar1.findOverlaps(date)
            获取 calendar2 的重叠情况 overlaps2 = calendar2.findOverlaps(date)
            对于 overlaps1 中的每个事件对 (event1, event2)
                对于 overlaps2 中的每个事件对 (event3, event4)
                    如果事件对时间范围重叠
                        将 (event1, event3) 或 (event2, event4) 添加到 allOverlaps
    返回 allOverlaps
结束
```

3. 日历的合并

该操作将一个日历的所有活动合并到另一个日历，并删除原日历。

```
函数 mergeCalendars 在 User 类中:
输入: sourceCalendarId: String, targetCalendarId: String
输出: void

开始
    查找 sourceCalendar = findCalendarById(sourceCalendarId)
    查找 targetCalendar = findCalendarById(targetCalendarId)
    对于 sourceCalendar.events 中的每个事件 event
        调用 targetCalendar.addEvent(event)
    调用 removeCalendar(sourceCalendarId)
结束
```

4. 计算用户某天的忙碌度

该操作计算用户在所有日历上指定日期的忙碌度。

```
函数 calculateBusyness 在 User 类中:
输入: date: Date
输出: double

开始
    设置 totalDuration = 0
    对于 user.calendars 中的每个 calendar
        对于 calendar.events 中的每个事件 event
            如果 event.date 等于 date
                计算事件持续时间 duration = event.endTime - event.startTime
                totalDuration += duration
    返回 totalDuration
结束
```

二、UML部署图

1)部署图基本方法

1. 确定要展示的元素

- **节点**：确定所有的物理资源（如服务器、工作站、移动设备等）和逻辑资源（如虚拟机、容器等）。
- **构件**：识别所有的软件构件，如应用程序、服务、数据库等，这些构件将部署在节点上。

2. 描述节点与构件的关系

- **节点内构件**：将构件放置在其部署的节点内部。这表示构件在该节点上运行。
- **节点间关系**：标示节点之间的通信或连接方式，如网络链接、数据传输路径等。

3. 使用标准符号

- **节点符号**：通常使用立方体来表示，可以标注节点名称。
- **构件符号**：使用矩形，并可能内嵌在节点符号内，标明名称和类型。
- **关系**：使用带箭头的线表示控制流或数据流，无箭头的线表示关联关系。

4. 明确显示依赖与流向

- **依赖关系**：显示构件或节点之间的依赖关系，如一个应用程序依赖特定的数据库服务。
- **数据流**：用箭头清楚标示数据流的方向，这有助于理解系统的运作方式。

5. 分层和分组

- 对节点和构件进行逻辑分组，如按功能、部门或物理位置等，可以使用包或分区来组织。
- 层次化显示可以帮助更清楚地展示复杂系统的结构，如将网络设备、服务器、应用层次清楚分开。

2)2015期末：5个需求的部署策略

1. 多平台支持与业务逻辑复用

策略：采用**微服务架构**，将业务逻辑封装在后端服务中，通过RESTful API提供服务。前端（Web, iOS, Android）只负责展示和交互，通过API与后端交互，实现业务逻辑的复用。

部署模型：

- 一个中心API服务器，负责处理来自各个客户端的请求。
- 分别为Web, iOS, Android开发不同的客户端应用，这些应用通过HTTP/HTTPS请求与API服务器交互。
- 使用容器技术（如Docker）部署后端服务，确保在不同环境中的一致性和可移植性。

2. 数据获取的效率和实时性

策略：采用**事件驱动的数据拉取机制**，监测站数据的更新触发通知到中心服务器，服务器再通过API拉取最新数据。同时，使用缓存技术减少对外部系统的直接请求，降低系统负担。

部署模型：

- 设置数据中心，负责存储缓存的监测站数据。
- 数据中心与外部监测站之间通过Webhook或消息队列（如Kafka）实现数据的实时更新。
- API服务器定期从数据中心拉取数据，确保数据的实时性和准确性。

3. 第三方信息的集成

策略：建立与交通、民航等部门的API集成，实时获取相关信息，并通过系统推送给用户。

部署模型：

- 与第三方部门的API接口进行集成。
- 使用API网关管理各种第三方服务的调用和数据整合。

4. 消息推送

策略：使用短信网关和推送通知服务（如Firebase Cloud Messaging for Android and Apple Push Notification Service for iOS）来推送消息给用户。

部署模型：

- 集成短信服务平台和移动推送服务。
- 通过批处理或事件驱动机制在特定条件下（如空气质量突变）触发消息推送。
- **消息推送系统**（Messaging System）包括短信网关、Firebase Cloud Messaging（FCM）、Apple Push Notification Service（APNS），用于向用户发送通知。

5. 弹性伸缩性

策略：采用云服务提供弹性伸缩的计算资源，根据访问量自动调整资源配置，确保系统在访问高峰期的性能，同时在空闲时节省资源。

部署模型：

- 使用云服务（如AWS EC2）自动扩展服务器实例。
- 部署负载均衡器（如Nginx或AWS ELB），均衡请求，优化性能。

2015期末思考题

1.软件估算和迭代开发

估算在迭代开发中的角色

1. **灵活性与适应性**：
   - 在迭代开发中，项目经常面对需求的变更和调整。估算提供了一个初始框架，帮助项目团队判断新需求或改变的可行性和对时间线的影响。
2. **资源分配**：
   - 虽然迭代开发具有高度的灵活性，但资源（如开发人员时间、资金等）往往是有限的。有效的估算帮助团队合理分配资源，优先处理最重要或最紧迫的任务。
3. **持续改进的基础**：
   - 迭代开发模型中的每一次迭代都可以视为一个学习和调整的机会。通过评估每个迭代的估算准确性和实际结果，团队可以不断优化他们的估算技巧和开发过程。
4. **透明性与沟通**：
   - 在敏捷开发中，估算过程（如规划扑克）也是团队协作和沟通的一部分。它帮助团队成员理解任务的复杂性和挑战，促进了对项目进展的共同理解。

5. **客户与利益相关者的期望管理**：
    - 即使在迭代开发中，客户和利益相关者仍然需要大概了解项目完成所需的时间和成本。有效的估算使得项目团队可以更好地设定这些外部期望。

如何有效地进行估算

1. **使用敏捷特有的估算方法**：
    - 如使用故事点来估算工作量而非直接用时间估算，这有助于考虑任务的复杂性、不确定性和努力程度。
2. **定期复审和调整估算**：
    - 在每个迭代结束时回顾估算的准确性，并根据最新的项目情况调整未来的估算。
3. **包括团队所有成员进行估算**：
    - 采用全员参与的估算方法（如规划扑克），确保估算反映了团队的集体经验和知识。

因此，即使在迭代开发中，估算依然是一个至关重要的活动。正确的估算不仅有助于项目管理和计划，还有助于团队内部的协作和沟通，以及对外部利益相关者的期望管理。正确理解和执行估算，可以显著提高迭代开发的效果和效率。

2.数据封装和public类型

数据封装是面向对象编程中的一个核心概念，其主要目的是确保对象的数据属性可以被隐藏起来，仅允许通过对象自身提供的方法来访问和修改这些数据。这样做的目的是提高模块的独立性和安全性，减少外部干扰，确保数据的完整性和一致性。

数据封装的基本理念

在严格遵循数据封装原则的情况下，确实，类中的属性应该不对外直接公开，这意味着这些属性不应该被设置为 `public`。这些属性通常被设置为 `private`，有时候也可以是 `protected`（当你希望在派生类中也能访问这些属性时）。

使用 `public` 属性的情况

然而，在实际编程中，将属性设置为 `public` 是允许的。这通常是出于以下几种考虑：

1. **简化代码**：在某些简单的或者不太关注封装的情况下，开发者可能会为了编码的便捷而将属性设为 `public`。
2. **数据结构类**：在一些使用面向对象技术实现的数据结构中，如链表节点、树节点等，属性可能会被设为 `public` 以便于从结构中的其他部分直接访问。
3. **性能考虑**：虽然这种情况较少，但是在某些极端性能敏感的应用中，直接访问属性而非通过方法访问可能略微提高效率。

推荐的做法

即使语言允许将属性设置为 `public`，遵循数据封装的最佳实践通常是推荐的。这样做有以下好处：

- **维护性**：封装使得类的实现细节可以独立于类的行为。这意味着类的内部实现可以随着需求变化而变化，而不会影响到使用该类的代码。
- **可控性**：通过提供方法（如 getter 和 setter）来访问和修改属性，可以在方法内部加入逻辑处理，比如检查数据有效性，或者触发事件等。
- **安全性**：隐藏属性的细节可以防止外部代码不当地修改内部状态，从而可能引起的错误或不一致。

结论

虽然技术上可以将类属性设为 `public`，但如果要严格遵守面向对象的数据封装原则，应尽量避免这样做。正确的做法是使用 `private` 或 `protected` 访问修饰符，并通过方法（如 getters 和 setters）来控制对这些属性的访问，从而更好地保护和管理对象的状态

3.细化的目的

在软件开发的详细设计阶段，对类之间的关联关系进行细化是一个关键步骤。这种细化的主要目的是确保系统的结构清晰、可维护，并且具备高效的实现逻辑。以下是对类关系细化的几个具体目的：

1. 明确类的责任和行为

细化类之间的关联关系有助于明确每个类的职责。通过定义类如何相互交互，可以确保每个类仅承担其应有的责任，遵循单一职责原则。这种明确的责任划分有助于减少类的复杂度，并增加代码的可读性和可维护性。

2. 提高系统的灵活性和可扩展性

当类之间的关系被恰当地定义和管理时，系统的整体结构会更加灵活，更容易应对需求变更。例如，通过使用接口或抽象类可以在不改变现有代码的基础上引入新的功能或者替换组件，从而提高了系统的可扩展性。

3. 优化数据流和控制流

通过细化类之间的关联关系，可以优化数据和控制流。明确哪些类负责数据的创建、哪些类负责数据的消费、哪些类提供控制逻辑等，有助于构建高效和合理的系统架构。这样的架构更能确保数据和控制流的准确性和高效性。

4. 降低耦合度

细化关联关系有助于实现类之间的低耦合度。通过定义清晰的接口和避免不必要的直接依赖，可以使各个类或组件更加独立，从而降低系统的整体复杂度，提高模块化水平。

5. 促进团队协作

在详细设计阶段清楚地定义和细化类之间的关系，有助于团队成员理解整个系统的设计和工作机制。这种清晰的设计文档和模型使得团队成员可以更容易地协作开发，尤其是在较大或分布式的团队中。

6. 减少实现过程中的错误

细化设计有助于在开发早期发现潜在的设计问题，从而避免在编码和后续测试阶段出现成本高昂的错误。预先定义好的类关系和交互逻辑可以减少实现时的歧义和误解。

4.状态图初始状态

1. 全局初始状态与局部初始状态

在状态图模型中区分全局初始状态和局部初始状态是理解这个问题的关键：

- **全局初始状态**：这是整个状态机开始执行时的起始点。整个状态机在开始时只能从这一个状态开始。

- **局部初始状态**：当状态机包含组合状态时，每个组合状态可以有自己的初始状态。这是进入该组合状态时默认进入的状态。

2. 组合状态的独立性

每个组合状态本质上是一个独立的子状态机，它内部的运作逻辑与外部状态机是隔离的。当外部状态机的转移导致进入某个组合状态时，该组合状态内部的初始状态就会被激活，而这并不影响外部状态机的全局初始状态的定义。

3. 状态机的层次结构

组合状态的使用引入了状态机的层次结构，这种层次结构使得状态机设计更加模块化和可管理。在这种设计中，每个层级的状态机（无论是顶层还是内嵌的组合状态）都可以有自己的初始状态，这些初始状态只在它们各自的上下文中有效。

4. 逻辑一致性

虽然在视觉上看起来可能有多个初始状态，但从逻辑上讲，整个状态机在任何时候都只能从一个全局初始状态开始执行。进入任何组合状态的操作都是由外部状态机的某个状态的转移决定的，而不是随意开始的。这保证了状态机的行为是一致和可预测的。

总结

因此，这两点并不矛盾。在状态机模型中，允许每个组合状态拥有自己的初始状态是为了保持子状态机的独立性和增强模型的清晰度与可维护性。这样的设计使得状态机可以在复杂系统中更有效地应用，同时保持了逻辑的清晰和一致性。

5.各类UML模型的重要性

1. 各种UML模型的作用

- **用例模型（Use Case Diagram）**：帮助开发者和利益相关者理解系统功能及用户交互方式，是需求捕捉和确认的关键工具。
- **类模型（Class Diagram）**：描述系统中的类及其属性、操作和类之间的关系，直接指导编程实现。
- **序列模型（Sequence Diagram）**：展示对象之间如何交互以及交互的顺序，对理解系统操作的具体实现过程非常有帮助。
- **状态模型（State Diagram）**：描述对象可能处于的各种状态以及在不同事件影响下的状态转换，对于设计复杂行为系统至关重要。
- **部署模型（Deployment Diagram）**：展示系统如何在物理设备上部署，包括硬件和网络架构等，对于系统的部署和维护阶段非常重要。

2. 模型的相互依赖性

各种UML模型通常不是孤立存在的，它们相互依赖，相互补充。例如：

- 类模型需要用例模型来定义需要实现的功能。
- 序列模型和状态模型可以帮助明确类模型中方法的实现细节。
- 部署模型依赖类模型来确定需要部署哪些软件组件。

3. 整体设计与局部实现的关系

虽然类模型直接关联到代码实现，但其他模型提供了不可或缺的上下文信息和设计验证，帮助确保开发的软件系统符合用户需求和业务目标。忽略这些模型可能导致开发的系统功能偏离用户实际需求，或者系统架构不适合部署和维护。

4. 整体软件开发流程的考量

在实际的软件开发过程中，各种模型的重要性会根据项目的具体需求、团队的工作流程和项目的复杂度而变化。在某些敏捷开发环境中，可能更侧重于用例和类模型，而在一些需要严格验证和维护的大型系统开发项目中，所有类型的模型都极为重要。

结论

因此，不能说类模型的重要性高于其他模型，或者可以忽略其他模型。合理使用各种UML模型能够更全面地支持软件开发的整个生命周期，从需求分析、系统设计到实现、部署和维护。各种模型共同协作，帮助团队构建符合需求、可维护且高效的软件系统。

2014期末思考题

1.敏捷开发

1. 敏捷强调团队协作而非个人全才

敏捷开发方法更多地强调团队协作和多功能团队的重要性，而不是单一成员的全面技能。在敏捷团队中，虽然团队成员通常被鼓励掌握多种技能（即“T型”技能模型，深度专业技能加上广泛的辅助技能），但并不要求每个成员都必须在所有技术领域中都具备高水平。团队的多样性——成员具有不同的专长和背景——可以增强团队的整体能力，促进更有效的问题解决。

2. 敏捷强调的是快速反应和适应变化

敏捷方法论的核心在于快速响应市场变化和客户需求，以及持续改进。敏捷团队需要的是能够快速学习、适应变化并寻找解决方案的能力。这并不完全依赖于技术水平的高低，而是依赖于团队的沟通、协作和问题解决能力。

3. 敏捷鼓励持续学习和成长

敏捷开发不只是关注现有的技术水平，它还鼓励持续学习和个人成长。通过定期的回顾和反思（如冲刺回顾），团队成员被鼓励识别改进领域并采取行动以增强个人和团队的能力。这种持续的改进文化使得团队即使在技术能力开始时不是顶尖水平，也能逐渐提升并适应敏捷开发的要求。

4. 敏捷团队结构支持多样技能组合

在敏捷团队中，常见的是跨职能的团队设置，团队成员可能包括开发人员、测试人员、UI/UX设计师等，他们共同协作完成产品的开发。这种结构允许团队成员利用各自的强项，同时在其他成员的帮助下补充自己的短板。

结论

敏捷开发不要求每个团队成员都是技术全才，而是需要团队成员能够有效协作、持续学习和适应变化。一个多样化的团队，拥有良好的沟通和协作能力，通常比一个技术能力很高但缺乏协作的团队更能体现敏捷开发的优势。因此，关键在于构建一个能力互补、持续进步的团队，而不是单纯追求技术水平的高低。

2.延期项目

Fredrick Brooks 在《人月神话》中提出的“Adding manpower to a late software project makes it later”（向一个已经延期的软件项目增加人力会使项目更加延期）的观点，基于他对软件开发项目的深入观察和经验总结。这个原则被称为“Brooks' Law”，在当今的软件开发实践中仍然具有相当的相关性，尽管其影响可能因采用的开发模式和管理策略的不同而有所变化。以下是该观点仍然成立的原因：

1. 沟通成本增加

增加项目团队的人数会导致沟通的复杂度增加。每增加一个团队成员，团队内部的沟通渠道就会成倍增加。在软件开发中，有效的沟通是非常重要的，沟通不畅会导致误解和错误，进而影响项目进度。

2. 协调成本上升

更多的团队成员意味着需要更多的管理和协调工作。项目经理需要花费更多时间在确保每个人都明白他们的角色和责任上，而不是专注于解决项目延期的实际问题。

3. 培训和上手时间

新加入的成员需要时间来了解项目的背景、架构和现有代码库。在他们变得生产效率之前，原有团队成员需要花时间培训新人，这本身就可能进一步推迟项目进度。

4. 产品复杂性和技术债务

软件项目往往具有高度复杂性，新团队成员可能很难快速掌握所有细节。此外，项目中可能存在的技术债务（如代码质量问题和设计缺陷）会使得新增人力难以快速产生效果。

当前软件开发模式下的变化

虽然Brooks' Law在很多情况下仍然适用，但现代软件开发实践，尤其是敏捷和精益开发方法，已经发展了一些策略来缓解这个问题：

- **迭代开发和持续集成**：敏捷开发鼓励小规模、跨职能的团队协作，通过短周期的迭代和频繁的集成来提高灵活性和响应变化的能力。
- **强调团队自治和自我组织**：现代开发团队倾向于自我管理，团队成员通常更加多才多艺，可以更快地适应和填补需要的角色。
- **工具和自动化的使用**：使用现代开发工具和自动化测试可以减少新成员的上手难度和减轻沟通负担。

3.SOLID

依赖倒置原则（DIP）

依赖倒置原则指出：

- 高层模块不应该依赖于低层模块。两者都应该依赖于抽象。
- 抽象不应该依赖于细节。细节应该依赖于抽象。

简而言之，这个原则鼓励程序员依赖于接口而不是具体实现，这样的设计可以减少组件间的耦合，增加系统的灵活性和可维护性。

开放封闭原则（OCP）

开放封闭原则表明：

- 软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。

这意味着设计应该能够在不修改现有代码的情况下进行扩展，从而允许系统随着需求变化而逐渐发展，而不必每次变更都进行大规模的重构。

它们之间的联系

- **共同的设计目标**：DIP 和 OCP 都旨在提高系统的灵活性和可维护性。通过依赖抽象（DIP）和允许系统扩展（OCP），这两个原则共同促进了设计的松耦合和模块间的独立性。
- **依赖抽象促进可扩展性**：DIP 通过促使高层和低层模块都依赖于抽象（通常是接口或抽象类），为遵循 OCP 铺平了道路。当系统需要新功能或行为时，可以通过添加新的具体实现来扩展系统，而无需修改现有的代码。
- **实现 OCP 的机制**：DIP 提供了一种机制，通过这种机制，新的功能可以被添加为新的模块或类，这些模块或类实现了现有的抽象。因此，系统可以轻松扩展而无需改变现有代码，从而实现了 OCP。

实际应用

在实际应用中，遵循 DIP 可以确保任何高层策略或业务逻辑都不直接依赖于低层的具体实现细节，而是依赖于抽象层。这样，当实现细节变化时，不会影响依赖它们的高层策略。遵循 OCP 的设计允许开发者通过扩展现有的抽象（添加新的派生类）来引入新功能，而不是通过修改现有的代码，这样可以降低维护成本，减少引入新错误的风险。

总结：五类原则之间联系

1. 单一职责原则（SRP）

- **定义**：一个类应该只有一个引起变化的原因。
- **联系**：SRP 保证了类的职责明确，这简化了类的测试和维护。它与其他原则关联，在减少代码依赖（通过 DIP）和容易扩展（通过 OCP）方面起着基础作用。

2. 开放封闭原则（OCP）

- **定义**：软件实体应当对扩展开放，对修改封闭。
- **联系**：OCP 促使设计者使用抽象和接口来允许系统在不修改现有代码的情况下进行扩展，这与 LSP 和 DIP 直接相关，因为这些原则通过多态和抽象使得扩展和修改更加容易且安全。

3. 里氏替换原则（LSP）

- **定义**：子类应该能够替换它们的基类而不影响程序的正确性。
- **联系**：LSP 支持 OCP 和 ISP 的实现，确保继承和抽象的正确使用，增强了程序的灵活性和可重用性。它保证了子类的行为符合父类的预期，这使得基于抽象编程（由 DIP 提倡）更加可靠。

4. 接口隔离原则（ISP）

- **定义**：客户端不应该依赖它不使用的接口。
- **联系**：ISP 推荐将大接口拆分成更小和更具体的接口，这样客户端只需要知道它们需要使用的方法。这降低了类之间的依赖关系（与 DIP 相关），同时也确保了类的设计遵循 SRP。

5. 依赖倒置原则（DIP）

- **定义**：高层模块不应依赖低层模块，两者应依赖抽象；抽象不应依赖细节，细节应依赖抽象。
- **联系**：DIP 是实现 OCP 的关键工具。通过依赖于抽象而非具体实现，高层模块的实现可以保持不变，即使低层模块经历了改动。同时，DIP 和 LSP 共同工作确保了通过抽象传递的行为是一致的。

综合联系

这些原则在一起提供了一个强大的框架，以支持面向对象设计的核心目标：创建可维护、可扩展且松耦合的系统。遵循 SOLID 原则的系统更容易理解、更简单测试，并能应对变化，从而降低了整体开发和维护的成本。通过使用这些原则，开发者可以避免代码的脆弱性和僵化性，使得软件项目更加健壮和适应未来的需求变化。

4.测试折中思想

1. **用例数量与覆盖度**：增加测试用例的数量可以提高覆盖度，有助于更全面地测试软件并发现更多潜在的错误。然而，这种增加往往伴随着成本和时间的上升，可能影响到测试效率。
2. **覆盖度与效率**：提高测试覆盖度通常需要复杂和详细的测试用例，这可能会降低测试的效率。实现高覆盖度而不牺牲效率需要精心设计的测试用例和可能的自动化测试支持。
3. **用例数量与效率**：测试用例数量的增加直接影响测试的执行时间和资源消耗。在有限的资源和时间条件下，增加测试用例数量可能会降低测试的整体效率。

在实际的软件测试过程中，测试团队需要在这三者之间找到合适的平衡点。这通常涉及到对项目的风险评估、重要功能的优先级分析以及资源的合理分配。有效的测试策略应当旨在最大化覆盖度和测试质量，同时考虑到成本和时间的约束，确保测试过程既全面又高效。使用自动化测试工具和优化的测试用例设计可以在这三者之间找到更好的平衡点。