深入解析 Node.js：事件循环与非阻塞 I/O

一、什么是Node.js？它是如何工作的？

1. 什么是Node.js

你可以将Node.js理解为**一个基于Chrome V8 JavaScript引擎构建的服务器端运行环境**，它并非编程语言（仍使用JavaScript），也非框架，核心作用是让JavaScript脱离浏览器环境，能够在服务器端执行文件读写、网络请求、数据库操作等后端开发任务。Node.js具有轻量、高效的特性，尤其适用于高并发的I/O密集型应用（如API接口、实时聊天、数据流处理等）。

2. Node.js的工作原理

1. **核心依赖**：基于Chrome V8引擎（高性能JavaScript解释器，将JS代码编译为机器码执行，保证运行效率），同时封装了libuv库（跨平台异步I/O库，提供事件循环、异步网络/文件操作等核心能力）。
2. **单线程+事件驱动**：Node.js主线程是单线程的，不会为每个请求创建新线程，而是通过**事件驱动模型**处理并发请求——所有异步操作（如文件读取、网络请求）都会被委托给底层线程池（由libuv管理），主线程仅负责监听事件完成后的回调函数，避免了线程切换的开销。
3. **执行流程**：JS代码由V8引擎解析执行，遇到异步I/O操作时，Node.js将其移交libuv线程池处理，主线程继续执行后续同步代码；当异步操作完成后，libuv会将对应的回调函数放入事件队列，等待事件循环取出并交由V8引擎执行。

二、Node.js中的全局对象有哪些？举例说明它们的用途

Node.js的全局对象无需通过 `require()` 导入，可在所有模块中直接访问，核心全局对象及用途如下：

1. **`global`**：Node.js的顶层全局对象（类似浏览器中的 `window`），所有全局变量（除了模块内的局部变量）都是 `global` 的属性。
   - 用途：定义全局可访问的变量或方法。
   - 示例：

```
Code block
// 在任意模块中定义全局变量
global.userName = "张三";
// 在其他模块中可直接访问
console.log(userName); // 输出：张三
```

2. `console`：提供控制台输出/调试功能，用于打印日志、错误信息等。
   - 用途：开发调试、日志记录。
   - 示例：

```
Code block
1  console.log("普通日志信息"); // 普通输出
2  console.error("错误信息提示"); // 错误输出（通常显示为红色）
3  console.table([{name: "张三"}, {name: "李四"}]); // 以表格形式输出数据
```

3. `process`：提供当前Node.js进程的信息和控制能力，是与操作系统交互的核心对象。
   - 用途：获取进程参数、环境变量、退出进程、监听进程事件等。
   - 示例：

```
Code block
1  console.log(process.pid); // 输出当前进程ID
2  console.log(process.env.NODE_ENV); // 输出环境变量（如开发环境/生产环境）
3  process.exit(0); // 正常退出进程（参数0表示正常退出，非0表示异常退出）
```

4. `Buffer`：用于处理二进制数据（Node.js原生不支持直接操作二进制数据，`Buffer`是核心解决方案）。
   - 用途：文件读写、网络数据传输、加密解密等场景中的二进制数据处理。
   - 示例：

```
Code block
1  // 创建一个Buffer实例
2  const buf = Buffer.from("Hello Node.js", "utf8");
3  console.log(buf.toString("utf8")); // 转换为字符串输出：Hello Node.js
4  console.log(buf.length); // 输出Buffer的字节长度
```

5. `setTimeout` / `setInterval` / `setImmediate`：定时器相关全局方法，用于延迟或周期性执行函数。
   - 用途：延迟执行任务、定时轮询、立即执行异步任务等。
   - 示例：

```
Code block
1  // 1秒后执行回调
2  setTimeout(() => console.log("1秒后执行"), 1000);
```

```
// 每2秒执行一次回调
setInterval(() => console.log("每2秒执行一次"), 2000);
// 事件循环下一轮立即执行
setImmediate(() => console.log("立即异步执行"));
```

三、Node.js的事件循环机制，它如何处理异步操作?

1. 事件循环（Event Loop）的核心定义

事件循环是Node.js实现**非阻塞异步编程的核心机制**，它是一个无限循环的流程，负责监听事件队列、取出异步操作完成后的回调函数，并将其交由V8引擎主线程执行，从而让单线程的Node.js能够处理大量并发异步请求。

2. 事件循环的阶段划分（从内到外，按执行优先级排序）

Node.js的事件循环分为6个阶段，每个阶段都会处理对应的回调队列，只有当前阶段的队列处理完毕（或达到最大执行数），才会进入下一个阶段：

1. **timers（定时器阶段）**：处理 `setTimeout` 和 `setInterval` 的回调函数（延迟时间到期后的回调）。
2. **pending callbacks（待定回调阶段）**：处理少数系统级回调（如TCP连接错误的回调）。
3. **idle, prepare（空闲/准备阶段）**：内部使用，开发者无需关注。
4. **poll（轮询阶段）**：核心阶段，负责监听I/O事件（文件I/O、网络I/O等），并处理对应的回调；若没有待处理的回调，会在此阶段阻塞等待新的I/O事件。
5. **check（检查阶段）**：处理 `setImmediate` 方法注册的回调函数。
6. **close callbacks（关闭回调阶段）**：处理关闭事件的回调（如 `socket.on('close', callback)`）。

3. 事件循环处理异步操作的流程

1. **同步代码优先执行**：Node.js启动后，先执行当前模块的同步代码，遇到异步操作（如 `setTimeout`、文件读取），不会阻塞主线程，而是将异步任务委托给底层libuv线程池（或系统内核）处理，同时将回调函数注册到对应的事件队列中。
2. **进入事件循环**：同步代码执行完毕后，主线程进入事件循环，按照上述6个阶段的优先级，依次处理每个阶段的回调队列。
3. **回调执行与循环往复**：每个阶段的回调队列处理完毕后，进入下一个阶段；当所有6个阶段处理完毕后，事件循环会再次从头开始（timers阶段），形成无限循环，直到没有待处理的回调和进程退出。
4. **微任务队列的优先级**：在每个事件循环阶段切换之前，会先处理所有**微任务队列**中的回调（Node.js中的微任务包括 `Promise.then/catch/finally`、`process.nextTick`），其中 `process.nextTick` 优先级高于普通Promise微任务。

示例（直观理解执行顺序）：

Code block

```
// 同步代码，优先执行
console.log("同步代码1");

// 微任务：process.nextTick
process.nextTick(() => console.log("process.nextTick 微任务"));

// 异步任务：setTimeout（timers阶段）
setTimeout(() => console.log("setTimeout 回调"), 0);

// 微任务：Promise
Promise.resolve().then(() => console.log("Promise 微任务"));

// 异步任务：setImmediate（check阶段）
setImmediate(() => console.log("setImmediate 回调"));

// 同步代码，其次执行
console.log("同步代码2");

// 输出顺序：
// 同步代码1
// 同步代码2
// process.nextTick 微任务
// Promise 微任务
// setTimeout 回调
// setImmediate 回调
```

四、Node.js中的非阻塞I/O（Non-blocking I/O）模型

1. 非阻塞I/O的核心定义

非阻塞I/O是Node.js的核心特性之一，指**当Node.js发起I/O操作（文件读取、网络请求、数据库操作等）时，主线程无需阻塞等待I/O操作完成，而是可以继续执行后续的同步代码，当I/O操作完成后，通过事件循环触发对应的回调函数进行处理**。

2. 非阻塞I/O与阻塞I/O的对比

- **阻塞I/O**：主线程发起I/O操作后，暂停执行所有代码，直到I/O操作完成（如传统的PHP文件读取），此时线程处于闲置状态，无法处理其他请求，并发性能低下。
- **非阻塞I/O**：主线程发起I/O操作后，立即返回一个状态（未完成），并继续执行后续代码；I/O操作由底层libuv线程池（或操作系统内核异步接口）在后台执行，完成后通过事件通知主线程，由事件循环处理回调，线程利用率极高。

3. 非阻塞I/O的实现依赖

1. **libuv库**：Node.js的底层依赖，负责封装跨平台的异步I/O接口，管理线程池（默认线程数为4，可通过 `UV_THREADPOOL_SIZE` 配置），将I/O操作委托给线程池处理，避免阻塞主线程。
2. **事件驱动模型**：非阻塞I/O的核心支撑，I/O操作完成后会触发对应的事件（如 `'data'`、`'end'`、`'close'`），回调函数被放入事件队列，等待事件循环取出执行。
3. **单线程主线程**：主线程仅负责执行同步代码和回调函数，不参与具体的I/O操作，从而保证了非阻塞的特性，同时避免了线程切换的开销。

示例（非阻塞文件读取）：

Code block

```
const fs = require('fs');

// 非阻塞I/O: fs.readFile（异步读取文件）
fs.readFile('test.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log("文件读取完成：", data);
});

// 主线程不阻塞，继续执行此代码
console.log("发起文件读取后，立即执行此同步代码");
```

五、如何在Node.js中实现模块化？require和exports是如何工作的？

Node.js采用**CommonJS模块化规范**（浏览器端常用ES6模块化，Node.js也支持ES6模块，需配置 `"type": "module"`），核心用于解决代码复用、作用域隔离、依赖管理等问题。

1. Node.js模块化的实现方式

Node.js中每个文件就是一个独立的模块，拥有独立的模块作用域（模块内的变量、方法默认仅在模块内部可访问，不会污染全局作用域），模块化的核心是「导出模块内容」和「导入模块内容」。

2. `exports` / `module.exports`：模块导出

用于将模块内的变量、方法、对象暴露出去，供其他模块导入使用，两者本质上是同一个引用（初期），但存在细微区别：

- **`exports`**：是 `module.exports` 的一个引用（快捷方式），默认是一个空对象 `{}`，只能通过添加属性/方法的方式导出内容，不能直接赋值（直接赋值会断开与 `module.exports` 的引用关联）。

示例：

Code block

```
// 模块A：a.js
// 导出单个属性
exports.name = "Node.js模块化";
// 导出方法
exports.sayHello = function() {
  console.log("Hello CommonJS");
};
```

- **`module.exports`**：是模块的真正导出对象，Node.js最终导出的是`module.exports`的值，而非`exports`。它既可以添加属性/方法，也可以直接赋值（如导出一个函数、类、非对象类型）。

示例：

Code block

```
// 模块A：a.js
// 方式1：添加属性（与exports效果一致）
module.exports.age = 10;
// 方式2：直接赋值（导出一个函数，此时exports失效）
module.exports = function() {
  return "直接导出一个函数";
};
```

注意：若同时使用`exports`添加属性和`module.exports`直接赋值，最终导出结果以`module.exports`为准（因为`module.exports`的赋值会覆盖原有引用）。

3. `require()`：模块导入

用于导入其他模块导出的内容，返回值是目标模块的`module.exports`对象，其工作流程如下：

1. **模块查找机制**：`require()`接收模块路径（相对路径、绝对路径、核心模块名、第三方模块名），按以下优先级查找：
   - 优先查找核心模块（如`fs`、`path`、`http`，Node.js内置，无需安装）；
   - 若为相对路径（`./`、`../`）或绝对路径，直接按路径查找对应的`.js`、`.json`、`.node`文件（可省略后缀，Node.js会自动补全）；
   - 若为第三方模块（如`express`），会在当前目录的`node_modules`文件夹中查找，若找不到则向上级目录的`node_modules`递归查找，直到根目录；
2. **模块缓存机制**：首次导入某个模块时，Node.js会执行该模块的代码，并将模块的`module.exports`对象缓存到`require.cache`中；后续再次导入该模块时，不会重新执行模块代码，直接从缓存中取出`module.exports`对象返回，提升性能。

3. **导入执行**：`require()` 导入模块时，会同步执行目标模块的所有代码（仅首次导入时执行），然后返回 `module.exports` 对象。

示例（模块导入）：

Code block

```
// 模块B：b.js
// 导入模块A
const moduleA = require('./a.js');

// 使用模块A导出的内容
console.log(moduleA.name); // 输出：Node.js模块化
moduleA.sayHello(); // 输出：Hello CommonJS

// 若模块A直接导出函数
const fn = require('./a.js');
console.log(fn()); // 输出：直接导出一个函数
```

4. 核心总结

- 每个文件=一个模块，拥有独立作用域；
- 导出：`exports` 是 `module.exports` 的引用，最终导出以 `module.exports` 为准；
- 导入：`require()` 按特定规则查找模块，缓存模块结果，返回 `module.exports` 对象；
- 执行特性：模块仅在首次导入时执行一次，后续导入直接使用缓存。

总结

1. Node.js是基于V8引擎的服务器端JS运行环境，通过单线程+事件驱动+libuv实现高效并发；
2. 核心全局对象有 `global`、`console`、`process`、`Buffer` 等，各有明确的实用场景；
3. 事件循环是异步核心，分6个阶段执行回调，微任务优先级高于宏任务；
4. 非阻塞I/O让主线程不等待I/O操作，通过线程池+事件循环处理异步回调；
1. Node.js默认使用CommonJS模块化，`exports/module.exports` 负责导出，`require()` 负责导入（带缓存和查找机制）。

六、从源码级别解释Node.js的模块加载机制

1. 模块加载核心入口：Module类

Node.js源码中，模块加载的核心逻辑封装在 `lib/module.js` 的 `Module` 类中，每个模块都是 `Module` 的实例。核心属性包括：`id`（模块唯一标识）、`exports`（模块导出对象）、`parent`（引入当前模块的父模块）、`filename`（模块文件路径）、`loaded`（模块是否加载完成）。

2. 源码层面的模块加载流程

从源码角度，`require()`触发的模块加载流程可分为5个核心步骤，对应源码中的关键方法：

1. **`Module._resolveFilename`步骤1：模块路径解析（）**
   源码中通过`Module._resolveFilename(request, parent, isMain)`方法解析模块的真实文件路径。核心逻辑：
   - 优先处理核心模块（如`fs`、`path`），直接返回内置模块路径；
   - 若为相对/绝对路径，结合父模块路径（`parent.filename`）拼接为完整路径，补全文件后缀（.js、.json、.node）；
   - 若为第三方模块，调用`lib/path.js`的路径查找逻辑，从当前目录`node_modules`开始递归向上查找，直到找到模块或抵达根目录；
   - 解析过程中会处理符号链接（`fs.realpathSync`），确保路径真实有效。

2. **`Module._cache`步骤2：模块缓存检查（）**
   Node.js源码中通过全局对象`Module._cache`（一个对象，key为模块真实路径，value为Module实例）缓存已加载的模块。
   - 解析出真实路径后，先检查`Module._cache`是否存在该模块实例；
   - 若存在，直接返回缓存的`module.exports`，避免重复加载；
   - 若不存在，创建新的`Module`实例，存入缓存（提前缓存，防止循环依赖时重复创建）。

3. **`Module.load`步骤3：模块加载（）**
   调用`module.load(filename)`方法加载模块，核心是根据文件类型选择对应的加载器（源码中定义在`Module._extensions`对象中）：
   - `.js`文件：通过`fs.readFileSync`同步读取文件内容，调用`Module._compile`编译执行；
   - `.json`文件：读取内容后通过`JSON.parse`解析为对象，赋值给`module.exports`；
   - `.node`文件：调用C++扩展模块的加载逻辑（`process.dlopen`），加载二进制模块。

4. **`Module._compile` 步骤4：模块编译执行（）**
   这是JS模块加载的核心步骤，源码中 `Module._compile(code, filename)` 负责将JS代码编译为可执行代码并执行。核心逻辑：
   - 封装代码：将模块原始代码包裹在一个匿名函数中，函数参数为 `exports` 、 `require` 、 `module` 、 `__filename` 、 `__dirname` ，确保模块作用域隔离（避免污染全局）；
   封装后的代码格式示例（源码动态拼接字符串）：
   ```
   (function (exports, require, module, __filename, __dirname) {
     // 模块原始代码
     const fs = require('fs');
     exports.name = 'test';
   });
   ```
   - 执行函数：通过 `vm.runInThisContext` （Node.js的虚拟机模块）执行封装后的函数，传入当前模块的 `exports` 、 `require` 等参数；
   - 执行过程中，模块内对 `exports` 或 `module.exports` 的修改会直接作用于Module实例的对应属性。
5. **`module.exports` 步骤5：返回导出对象（）**
   模块编译执行完成后，将 `module.exports` 作为 `require()` 的返回值，供父模块使用。

3. 循环依赖的源码处理逻辑

当A模块依赖B模块，B模块又依赖A模块时，源码通过“提前缓存+部分导出”解决：
- A模块加载时，创建A的Module实例并存入缓存，然后加载B模块；
- B模块加载时依赖A，从缓存中取出A的实例（此时A尚未执行完成， `module.exports` 可能是初始空对象）；
- B模块执行完成后导出自己的 `exports` ，A模块继续执行，补全自己的 `module.exports` ；
- 源码中通过 `module.loaded` 属性标记模块是否执行完成，确保循环依赖时不会死锁。

七、什么是中间件 (middleware)？在Express、Koa中如何使用中间件？

1. 中间件的核心定义

中间件是Web框架（如Express、Koa）中用于处理HTTP请求的“管道式”组件，本质是一个函数。它可以：
- 接收请求对象（ `req` ）、响应对象（ `res` ）、后续中间件的回调函数（Express）或上下文对象（ `ctx` ）、next函数（Koa）；
- 对请求进行预处理（如解析参数、验证权限、记录日志）；
- 终止请求-响应循环（如直接返回响应）；
- 调用下一个中间件（通过 `next()` ），形成处理链条。

2. Express中间件的使用

Express中间件的核心格式：`(req, res, next) => { ... }`，其中`next()`用于调用下一个中间件。支持4种类型的中间件，使用方式如下：

1. **应用级中间件（全局/路由前缀）**
   作用于整个应用或指定路由前缀，通过`app.use()`注册。
   示例：

```
const express = require('express');
const app = express();

// 全局应用级中间件：记录所有请求日志
app.use((req, res, next) => {
  console.log(`[${new Date().toLocaleString()}] ${req.method} ${req.url}`);
  next(); // 调用下一个中间件
});

// 带路由前缀的应用级中间件：仅作用于/api开头的路由
app.use('/api', (req, res, next) => {
  console.log("处理/api前缀的请求");
  next();
});

// 路由处理
app.get('/api/user', (req, res) => {
  res.send("用户列表");
});

app.listen(3000);
```

2. **路由级中间件**
   作用于特定路由，通过 `router.use()` 或直接在路由定义中注册。
   示例：

```
const router = express.Router();

// 路由级中间件：仅作用于当前router的所有路由
router.use((req, res, next) => {
  console.log("路由级中间件处理");
  next();
});

// 直接在路由中注册中间件（多个中间件可传入数组）
router.get('/detail', (req, res, next) => {
  req.userInfo = { id: 1, name: "张三" }; // 预处理数据
  next();
}, (req, res) => {
  res.send(`用户信息：${JSON.stringify(req.userInfo)}`);
});

app.use('/user', router);
```

3. **错误处理中间件**
   专门处理请求过程中的错误，格式为 `(err, req, res, next) => { ... }`（比普通中间件多一个 `err` 参数），需通过 `app.use()` 注册，且必须放在所有中间件和路由之后。
   示例：

```
app.use((err, req, res, next) => {
  console.error("错误信息：", err.stack);
  res.status(500).send("服务器内部错误");
});

// 触发错误：在中间件或路由中调用next(err)
app.get('/error', (req, res, next) => {
  const err = new Error("自定义错误");
  next(err); // 交给错误处理中间件
});
```

4. **内置/第三方中间件**
   Express内置了部分中间件（如`express.json()`解析JSON请求体、`express.urlencoded()`解析表单数据），第三方中间件需安装后使用（如`cors`处理跨域）。
   示例：
   `// 内置中间件：解析JSON请求体`
   `app.use(express.json());`
   `// 内置中间件：解析x-www-form-urlencoded表单数据`
   `app.use(express.urlencoded({ extended: false }));`

   `// 第三方中间件：处理跨域`
   `const cors = require('cors');`
   `app.use(cors());`

3. Koa中间件的使用

Koa（基于ES6+）的中间件核心格式：`(ctx, next) => { ... }`，其中`ctx`是上下文对象（封装了`req`和`res`），`next()`是返回Promise的函数，支持异步（async/await）。Koa中间件采用“洋葱模型”执行，使用方式如下：

1. **全局中间件**

   通过`app.use()`注册，作用于所有请求，支持异步处理。

   示例（洋葱模型演示）：

```
const Koa = require('koa');
const app = new Koa();

// 中间件1：异步中间件
app.use(async (ctx, next) => {
  console.log("中间件1 开始");
  await next(); // 等待下一个中间件执行完成
  console.log("中间件1 结束");
});

// 中间件2
app.use(async (ctx, next) => {
  console.log("中间件2 开始");
  ctx.body = "Hello Koa"; // 设置响应体
  await next();
  console.log("中间件2 结束");
});

// 中间件3
app.use(async (ctx, next) => {
  console.log("中间件3 开始");
  await next();
  console.log("中间件3 结束");
});

// 输出顺序：中间件1 开始 > 中间件2 开始 > 中间件3 开始 > 中间件3 结束 > 中间件2 结束 > 中间件1 结束
app.listen(3000);
```

2. **路由级中间件**

Koa本身无内置路由，需使用第三方包`koa-router`，路由级中间件通过`router.use()`或路由定义中注册。

示例：

```
const Router = require('koa-router');
const router = new Router({ prefix: '/api' }); // 路由前缀

// 路由级中间件：仅作用于当前路由
router.use(async (ctx, next) => {
  console.log("路由级中间件处理");
  if (!ctx.query.token) {
    ctx.status = 401;
    ctx.body = "缺少token";
    return;
  }
  await next();
});

router.get('/user', async (ctx) => {
  ctx.body = { id: 1, name: "张三" };
});

app.use(router.routes()).use(router.allowedMethods()); // 注册路由及允许的HTTP方法
```

3. **错误处理中间件**
   Koa的错误处理可通过封装中间件实现，结合 `try/catch` 捕获异步错误（因Koa支持async/await，同步错误和异步错误均可通过try/catch捕获）。
   示例：

```
// 全局错误处理中间件（需放在所有中间件最前面）
app.use(async (ctx, next) => {
  try {
    await next(); // 执行后续中间件
    // 处理404
    if (ctx.status === 404) {
      ctx.status = 404;
      ctx.body = "页面不存在";
    }
  } catch (err) {
    console.error("错误信息：", err.stack);
    ctx.status = err.status || 500;
    ctx.body = err.message || "服务器内部错误";
  }
});

// 触发错误
app.use(async (ctx, next) => {
  throw new Error("自定义错误");
});
```

八、Express、Koa的源码是如何实现和工作的?

1. Express源码实现核心

Express源码（核心文件：`lib/express.js`、`lib/application.js`、`lib/router/index.js`）核心是“中间件链表+路由匹配”，工作流程如下：

1. **核心类：Application**
   Express的入口是 `express()` 函数，该函数返回 `Application` 类的实例（`app`）。
   - `Application` 类继承自 `events.EventEmitter`，支持事件监听（如 `error` 事件）；
   - 核心属性：`_router`（Router实例，管理路由和中间件）、`settings`（应用配置）、`middleware`（应用级中间件数组）；
   - 核心方法：`use()`（注册中间件，最终调用 `_router.use()`）、`get()`/`post()` 等HTTP方法（注册路由，调用 `_router.METHOD()`）、`listen()`（启动HTTP服务器，调用 `http.createServer(this)`，并将 `app` 作为请求处理函数）。

2. **路由与中间件管理：Router类**
   `Router` 类是Express处理路由的核心，每个 `app` 实例内置一个 `_router` ，也可通过 `express.Router()` 创建独立路由。
   - 核心属性： `stack` （中间件/路由规则数组，每个元素是 `Layer` 实例，包含路径、中间件函数、匹配规则）；
   - `use()` 方法：向 `stack` 中添加中间件Layer，支持路径前缀匹配；
   - HTTP方法（如 `get()` ）：创建路由Layer，关联请求方法、路径和处理函数，存入 `stack` 。

3. **请求处理流程**
   当HTTP请求到达时，服务器调用 `app(req, res)` （ `Application` 类实现了 `req, res, next` 的请求处理函数），核心逻辑：
   1. 初始化 `req` （扩展原生req，如添加 `req.query` 、 `req.params` ）和 `res` （扩展原生res，如添加 `res.send()` 、 `res.json()` ）；
   2. 调用 `_router.handle(req, res, done)` ，启动路由匹配和中间件执行；
   3. `Router.handle()` 遍历 `stack` 中的Layer，通过 `path-to-regexp` 库匹配请求路径和方法；
   4. 匹配成功后，执行Layer中的中间件函数，通过 `next()` 控制流程（ `next()` 内部是递归遍历stack的逻辑）；
   5. 若所有中间件执行完成仍未返回响应，调用 `done()` ，触发404错误。

4. **响应方法实现**
   Express扩展了原生 `res` 对象，如 `res.send()` （源码在 `lib/response.js` ）：
   - 自动判断响应数据类型（字符串、JSON、Buffer），设置对应的 `Content-Type` ；
   - 处理响应状态码，调用原生 `res.end()` 发送响应。

2. Koa源码实现核心

Koa（源码核心文件： `lib/application.js` ）基于ES6+，设计更简洁，核心是“洋葱模型中间件+上下文封装”，工作流程如下：

1. **核心类：Application**
   `koa()` 返回 `Application` 实例，继承自 `events.EventEmitter` 。
   - 核心属性： `middleware` （中间件数组，存储所有通过 `use()` 注册的中间件函数）、 `context` （上下文对象原型，封装 `req` 、 `res` ）；
   - 核心方法：
     - `use(fn)` ：向 `middleware` 数组添加中间件（需是 `(ctx, next) => { ... }` 格式，若为Generator函数，需通过 `koa-convert` 转换）；
     - `listen()` ：创建HTTP服务器，调用 `http.createServer(this.callback())` ，其中 `this.callback()` 返回请求处理函数。

2. **上下文封装（ctx）**
   Koa通过`context`、`request`、`response`三个对象封装原生`req`和`res`：
   - `context`：作为中间件的第一个参数，通过`Object.create()`创建实例，原型链指向`app.context`；
   - `ctx.req`/`ctx.res`：原生req/res对象；
   - `ctx.request`/`ctx.response`：Koa扩展的请求/响应对象（如`ctx.request.url`、`ctx.response.body`）；
   - 源码中通过getter/setter简化操作，如`ctx.body`对应`ctx.response.body`，`ctx.status`对应`ctx.response.status`。

3. **洋葱模型中间件实现**
   Koa的核心是`compose`函数（源码在`lib/compose.js`），该函数接收中间件数组，返回一个可执行的中间件函数，实现洋葱模型的核心逻辑：
   - `compose`函数内部通过递归调用`next()`，将中间件串联起来；
   - 每个中间件的`next()`函数对应下一个中间件的执行结果（Promise）；
   - 当中间件中调用`await next()`时，会暂停当前中间件，执行下一个中间件，直到最内层中间件执行完成后，再反向执行每个中间件的后续逻辑。
   `compose`核心源码简化：

```
function compose(middleware) {
  return function (ctx, next) {
    let index = -1;
    function dispatch(i) {
      if (i <= index) return Promise.reject(new Error("next() called multiple times"));
      index = i;
      let fn = middleware[i];
      if (i === middleware.length) fn = next;
      if (!fn) return Promise.resolve();
      try {
        return Promise.resolve(fn(ctx, dispatch.bind(null, i + 1)));
      } catch (err) {
        return Promise.reject(err);
      }
    }
    return dispatch(0);
  };
}
```

4. **请求处理流程**
   1. 调用 `app.callback()`，内部通过 `compose(app.middleware)` 生成中间件执行函数 `fn`；
   2. 返回请求处理函数 `(req, res) => { ... }`，该函数创建 `ctx` 实例（`ctx = app.createContext(req, res)`）；
   3. 执行 `fn(ctx)`，启动中间件链条；
   4. 中间件执行完成后，根据 `ctx.body`、`ctx.status` 等信息，调用 `respond()` 函数发送响应（源码在 `lib/application.js`）；
   5. 若执行过程中出现错误，触发 `error` 事件，若未监听 `error` 事件，直接抛出错误。

九、什么是数据库？为什么需要数据库存储？

1. 数据库的核心定义

数据库（Database，简称DB）是**按照数据结构组织、存储和管理数据的仓库**，本质是一个或多个结构化的数据集合。它由数据库管理系统（Database Management System，简称DBMS）进行管理，DBMS提供数据定义、数据操作（增删改查）、数据安全、数据备份等核心功能。
简单来说，数据库就是“专业的数据管家”，负责高效、安全地存储和管理大量数据，供应用程序（如网站、APP）调用。

2. 需要数据库存储的核心原因

在没有数据库的时代，数据通常存储在文件中（如txt、excel），但随着数据量增长和应用复杂度提升，文件存储的弊端凸显，数据库的价值由此体现，核心原因如下：

1. **解决数据冗余问题**
   文件存储中，相同数据可能需要在多个文件中重复存储（如用户信息在订单文件、评论文件中都需保存），导致数据冗余（重复存储），浪费存储空间且难以维护（修改一处数据需同步修改多处）。
   数据库通过数据规范化设计（如关系型数据库的表结构设计），将数据拆分到不同表中，通过关联关系避免重复存储，降低冗余。

2. **保证数据一致性与完整性**
   文件存储中，数据修改容易出现“部分修改成功”的情况（如修改用户姓名，仅修改了订单文件中的姓名，评论文件中的姓名未修改），导致数据不一致。
   数据库通过约束（如主键约束、外键约束、唯一性约束）和事务（原子性、一致性、隔离性、持久性，ACID）保证数据完整性和一致性：
   - 约束：防止无效数据插入（如主键不能为空、唯一）；
   - 事务：确保一组数据操作要么全部成功，要么全部失败（如转账操作，扣款和到账必须同时完成）。

3. **支持高效的数据查询与统计**
   文件存储中，查询数据需要遍历整个文件（如从10万条订单中查询某用户的订单），效率极低；复杂查询（如按时间范围、金额区间统计订单）几乎无法实现。
   数据库提供结构化查询语言（SQL）或专用查询接口（如MongoDB的聚合查询），支持复杂条件查询、排序、分组、统计等操作，且通过索引优化查询性能，可快速从海量数据中定位目标数据。

4. **实现数据共享与并发访问**
   文件存储难以支持多用户同时访问（如多个用户同时修改同一文件，容易导致数据错乱），且无法实现不同应用程序共享数据。
   数据库支持多用户、多应用程序并发访问，通过锁机制（如行锁、表锁）避免并发冲突，同时提供数据共享接口，让多个应用程序可安全地访问同一数据库中的数据（如电商系统的订单模块和支付模块共享用户订单数据）。

5. **提升数据安全性**
   文件存储的安全防护能力弱，数据容易被篡改、删除或泄露（仅能依赖文件系统的权限控制）。
   数据库提供完善的安全机制：
   - 身份认证：用户需输入账号密码才能访问数据库；
   - 权限控制：可细粒度控制用户权限（如某用户仅能查询数据，不能修改数据）；
   - 数据备份与恢复：支持自动备份数据，可在数据丢失或损坏时快速恢复；
   - 数据加密：对敏感数据（如密码、手机号）进行加密存储，防止泄露。

6. **便于数据管理与维护**
   文件存储中，数据的新增、修改、删除需要手动操作文件，难以管理；数据备份、日志记录等维护工作繁琐。
   数据库管理系统（DBMS）提供可视化管理工具（如MySQL的Navicat、MongoDB的Compass）和命令行工具，支持数据批量操作、自动备份、日志监控等功能，降低数据管理和维护的成本。

十、常见的数据库有哪些?什么是关系型数据库和非关系型数据库? 它们有什么区别?

1. 常见的数据库

数据库按数据模型可分为关系型数据库和非关系型数据库，常见类型及代表如下：

1. **关系型数据库（RDBMS）**
   代表：MySQL（开源，应用最广泛）、Oracle（商业，高性能，适用于大型企业）、SQL Server（微软，适用于Windows环境）、PostgreSQL（开源，支持复杂数据类型和SQL标准）、SQLite（嵌入式，适用于轻量级应用，如移动端APP）。

2. **非关系型数据库（NoSQL）**
   代表：
   - 文档型：MongoDB（存储JSON格式文档，适用于内容管理、日志存储）；
   - 键值型：Redis（内存数据库，适用于缓存、会话存储、实时统计）；
   - 列族型：HBase（适用于海量数据存储，如大数据分析场景）；
   - 图数据库：Neo4j（适用于关系网络分析，如社交网络、推荐系统）。

2. 关系型数据库和非关系型数据库的定义

1. **关系型数据库（Relational Database）**
   基于关系模型（由二维表组成，表中的行是记录，列是字段）的数据库，数据之间通过主键、外键建立关联关系。核心特征：
   - 数据结构化：以表为单位存储数据，表结构固定（需提前定义字段名、数据类型、约束）；
   - 支持SQL：通过结构化查询语言（SQL）进行数据操作；
   - 遵循ACID事务：保证数据的原子性、一致性、隔离性、持久性；
   - 数据完整性：通过主键、外键、唯一性约束等保证数据有效。
2. **非关系型数据库（Non-Relational Database，简称NoSQL）**
   不依赖关系模型的数据库，数据存储格式灵活（如文档、键值对、列族、图），核心特征：
   - 数据非结构化/半结构化：无需提前定义数据结构，可动态添加字段（如MongoDB的文档可包含不同字段）；
   - 不依赖SQL：部分支持类SQL查询（如MongoDB的MongoSQL），但无统一查询标准；
   - 事务支持灵活：多数NoSQL数据库不支持完整ACID事务（如Redis仅支持简单事务，MongoDB 4.0+支持多文档事务），更强调最终一致性；
   - 高并发、高可扩展性：专为海量数据和高并发访问设计，支持分布式部署。

3. 关系型数据库和非关系型数据库的核心区别

对比维度	关系型数据库	非关系型数据库
数据模型	二维表结构，数据间有明确关联	文档、键值对、列族、图等，数据间关联弱
数据结构	固定（需提前定义表结构）	灵活（动态添加字段，无需预定义）
查询语言	支持标准SQL，查询功能强大	无统一标准，各数据库有专用查询接口（部分支持类SQL）
事务支持	支持完整ACID事务，数据一致性强	多数不支持完整ACID，支持最终一致性或部分事务（如Redis简单事务）
并发性能	适用于中低并发，高并发下需通过分库分表优化	高并发性能优异，支持分布式部署，可横向扩展

扩展性	纵向扩展（提升单服务器配置）为主，横向扩展复杂	横向扩展容易（增加服务器节点），支持分布式存储
数据完整性	通过主键、外键、约束等保证数据完整性	依赖应用程序逻辑保证数据完整性，数据库层面约束弱
适用场景	数据关系复杂、需要强一致性的场景（如电商订单、金融交易、用户管理）	海量数据存储、高并发访问、数据结构不固定的场景（如缓存、日志、社交网络、大数据分析）

4. 总结

关系型数据库适合数据结构固定、关系复杂、需要强一致性的场景，核心优势是数据完整性和事务支持；非关系型数据库适合数据量大、并发高、数据结构灵活的场景，核心优势是高可扩展性和高并发性能。实际开发中，两者常结合使用（如用MySQL存储核心业务数据，Redis做缓存提升访问速度，MongoDB存储日志数据）。

十一、MySQL的存储引擎有哪些？InnoDB与MyISAM的区别是什么？

1. MySQL常见的存储引擎

存储引擎是MySQL处理数据的核心组件，负责数据的存储、索引构建、事务管理等底层操作，MySQL支持多种存储引擎，可通过`SHOW ENGINES;`命令查看支持的引擎及状态。常见存储引擎如下：

1. **InnoDB**：MySQL 5.5及以上版本的默认存储引擎，支持事务、行级锁、外键约束，适用于需要强一致性的业务场景（如电商订单、金融交易）。
2. **MyISAM**：早期MySQL的默认存储引擎，不支持事务和外键，采用表级锁，查询性能优异，适用于只读或读写较少的场景（如静态网站数据、日志存储）。
3. **Memory**：数据存储在内存中，读写速度极快，但重启MySQL后数据丢失，适用于临时数据存储、缓存场景（如临时统计结果）。
4. **Archive**：专为归档设计，支持高压缩比，仅支持插入和查询操作，适用于海量历史数据归档（如多年前的用户日志）。
5. **CSV**：数据以CSV文件格式存储，可直接用文本编辑器查看，适用于数据导入导出场景（如与其他系统数据交互）。

2. InnoDB与MyISAM的核心区别

对比维度	InnoDB	MyISAM
事务支持	支持完整ACID事务（原子性、一致性、隔离性、持久	不支持事务

	性）	
锁机制	支持行级锁（默认）和表级锁，并发性能优异（仅锁定修改的行，不影响其他行）	仅支持表级锁（修改表中任意一行都会锁定整个表），并发性能差
外键约束	支持外键约束（保证数据完整性和一致性）	不支持外键约束
数据存储结构	数据和索引存储在同一个文件（.ibd），采用聚簇索引（索引与数据绑定）	数据文件（.MYD）和索引文件（.MYI）分离，采用非聚簇索引
崩溃恢复	支持崩溃恢复（通过事务日志（redo log、undo log）恢复数据，保证数据不丢失）	不支持崩溃恢复，数据可能丢失（仅记录数据修改的位置，无完整日志）
查询性能	读写混合场景、并发场景性能优，但单表查询速度略低于MyISAM	单表查询速度快（索引与数据分离，查询无需关联数据文件），适合只读场景
适用场景	电商订单、金融交易、用户管理等需要事务、高并发的核心业务	静态网站数据、日志存储、报表统计等只读或读写较少的场景

十二、MySQL中有哪些常见的操作?

MySQL的常见操作围绕“数据库管理”“表管理”“数据操作”“查询操作”“权限管理”展开，核心使用SQL语句实现，以下是分类整理的常见操作及示例：

1. 数据库管理操作

1. 创建数据库：`CREATE DATABASE IF NOT EXISTS 数据库名 DEFAULT CHARACTER SET utf8mb4;`（指定字符集为utf8mb4，支持表情符号）
2. 查看数据库：`SHOW DATABASES;`（查看所有数据库）、`USE 数据库名;`（切换到目标数据库）
3. 删除数据库：`DROP DATABASE IF EXISTS 数据库名;`（删除前判断是否存在，避免报错）
4. 查看数据库配置：`SHOW CREATE DATABASE 数据库名;`（查看数据库的创建语句及字符集配置）

2. 表管理操作

1. 创建表：`CREATE TABLE IF NOT EXISTS 表名 (字段名 数据类型 约束, ...);`
   示例（创建用户表）：
   ```
   CREATE TABLE IF NOT EXISTS user (
     id INT PRIMARY KEY AUTO_INCREMENT COMMENT '用户ID（主键，自增）',
     username VARCHAR(50) NOT NULL UNIQUE COMMENT '用户名（非空，唯一）',
     password VARCHAR(100) NOT NULL COMMENT '密码（加密存储）',
     age INT DEFAULT 0 COMMENT '年龄（默认0）',
     create_time DATETIME DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间（默认当前时间）'
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
   ```
2. 查看表：`SHOW TABLES;`（查看当前数据库所有表）、`DESC 表名;`（查看表结构）、`SHOW CREATE TABLE 表名;`（查看表创建语句）
3. 修改表：
   - 添加字段：`ALTER TABLE 表名 ADD 字段名 数据类型 约束;`
   - 修改字段：`ALTER TABLE 表名 MODIFY 字段名 新数据类型 新约束;`
   - 删除字段：`ALTER TABLE 表名 DROP 字段名;`
   - 修改表名：`ALTER TABLE 旧表名 RENAME TO 新表名;`
4. 删除表：`DROP TABLE IF EXISTS 表名;`

3. 数据操作（DML：数据操纵语言）

1. 插入数据：
   - 全字段插入：`INSERT INTO 表名 VALUES (值1, 值2, ...);`
   - 指定字段插入：`INSERT INTO 表名 (字段1, 字段2, ...) VALUES (值1, 值2, ...);`
   - 批量插入：`INSERT INTO 表名 (字段1, 字段2) VALUES (值1, 值2), (值3, 值4);`
2. 修改数据：`UPDATE 表名 SET 字段1=值1, 字段2=值2 WHERE 条件;`（必须加WHERE条件，否则修改全表数据）
   示例：`UPDATE user SET age=25 WHERE id=1;`
3. 删除数据：
   - 按条件删除：`DELETE FROM 表名 WHERE 条件;`
   - 清空表数据（不删除表结构，自增ID重置）：`TRUNCATE TABLE 表名;`（注意：TRUNCATE无法回滚，DELETE可以通过事务回滚）

4. 数据查询（DQL：数据查询语言）

核心语法：`SELECT 字段列表 FROM 表名 WHERE 条件 GROUP BY 分组字段 HAVING 分组条件 ORDER BY 排序字段 LIMIT 分页参数;`

1. 基础查询：
   - 查询所有字段：`SELECT * FROM 表名;`（不推荐，效率低，尽量指定字段）
   - 查询指定字段：`SELECT id, username FROM user;`
   - 去重查询：`SELECT DISTINCT age FROM user;`（去除重复的age值）
2. 条件查询：使用WHERE子句，支持=、!=、>、<、BETWEEN...AND、IN、LIKE、IS NULL等条件
   示例：
   - 查询年龄大于20的用户：`SELECT * FROM user WHERE age > 20;`
   - 查询年龄在18-30之间的用户：`SELECT * FROM user WHERE age BETWEEN 18 AND 30;`
   - 查询用户名包含“张”的用户：`SELECT * FROM user WHERE username LIKE '%张%';`（%表示任意字符）
3. 排序查询：`ORDER BY 字段 排序方式`（ASC升序，默认；DESC降序）
   示例：`SELECT * FROM user ORDER BY create_time DESC;`（按创建时间降序）
4. 分组查询：`GROUP BY 字段`，结合聚合函数（COUNT、SUM、AVG、MAX、MIN）使用
   示例：查询各年龄的用户数量：`SELECT age, COUNT(*) AS user_count FROM user GROUP BY age;`
5. 分页查询：`LIMIT 偏移量, 每页条数`（偏移量=（页码-1）*每页条数）
   示例：查询第2页数据，每页10条：`SELECT * FROM user LIMIT 10, 10;`
6. 多表关联查询：支持INNER JOIN（内连接，取交集）、LEFT JOIN（左连接，取左表所有数据）、RIGHT JOIN（右连接，取右表所有数据）
   示例（查询用户及对应的订单数据）：
   `SELECT u.username, o.order_id FROM user u INNER JOIN order o ON u.id = o.user_id;`

5. 权限管理操作

1. 创建用户：`CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';`（主机名用%表示任意主机）
2. 授权用户：`GRANT 权限列表 ON 数据库名.表名 TO '用户名'@'主机名';`
   示例：授予用户查询和修改user表的权限：`GRANT SELECT, UPDATE ON test.user TO 'test_user'@'%';`
3. 查看权限：`SHOW GRANTS FOR '用户名'@'主机名';`
4. 撤销权限：`REVOKE 权限列表 ON 数据库名.表名 FROM '用户名'@'主机名';`
5. 删除用户：`DROP USER '用户名'@'主机名';`

十三、什么是事务？MySQL中如何管理事务？

1. 事务的核心定义

事务（Transaction）是MySQL中一组不可分割的SQL操作集合，这组操作要么全部执行成功，要么全部执行失败（即“原子性”），不会出现部分成功、部分失败的情况。事务的核心作用是保证数据的一致性和完整性，尤其适用于多步操作的业务场景（如转账：扣款和到账两个操作必须同时完成）。

2. 事务的ACID特性（核心原则）

1. **原子性（Atomicity）**：事务中的所有操作是一个不可分割的整体，要么全部执行，要么全部回滚（撤销），不存在中间状态。例如：转账时，“用户A扣款”和“用户B到账”必须同时成功，若其中一步失败，整个事务回滚，数据恢复到初始状态。
2. **一致性（Consistency）**：事务执行前后，数据的完整性约束（如主键唯一、外键关联）不被破坏。例如：转账前A和B的总金额为1000元，转账后总金额仍为1000元。
3. **隔离性（Isolation）**：多个事务并发执行时，一个事务的执行不会被其他事务干扰，每个事务都感觉不到其他事务的存在。MySQL通过隔离级别控制并发事务的干扰程度。
4. **持久性（Durability）**：事务执行成功后，对数据的修改是永久性的，即使MySQL服务崩溃或服务器断电，数据也不会丢失（通过事务日志保证）。

3. MySQL中事务的管理方式

MySQL中只有InnoDB存储引擎支持事务，事务管理通过以下SQL语句实现，核心是“开启事务→执行SQL→提交/回滚事务”：

1. **开启事务**：
   - 方式1：`START TRANSACTION;`（推荐，清晰易懂）
   - 方式2：`BEGIN;`（与START TRANSACTION功能一致）
2. **执行事务内的SQL操作**：执行一组DML语句（INSERT、UPDATE、DELETE），这些语句是事务的核心操作。
   示例（转账事务）：
   `START TRANSACTION;`
   `// 1. 用户A扣款100元（A的余额减少100）`
   `UPDATE user SET balance = balance - 100 WHERE id = 1;`
   `// 2. 用户B到账100元（B的余额增加100）
UPDATE user SET balance = balance + 100 WHERE id = 2;`
3. **提交事务**：若所有SQL操作执行成功，通过`COMMIT;`提交事务，此时对数据的修改永久生效。
   示例：`COMMIT;`（转账成功，数据永久保存）
4. **回滚事务**：若其中任意一条SQL执行失败，通过`ROLLBACK;`回滚事务，所有操作的修改全部撤销，数据恢复到事务开启前的状态。
   示例：`ROLLBACK;`（转账失败，A和B的余额恢复初始值）

5. **保存点（可选）**：对于复杂事务，可通过保存点（Savepoint）实现部分回滚，无需回滚整个事务。
   - 创建保存点：`SAVEPOINT 保存点名称;`
   - 回滚到保存点：`ROLLBACK TO 保存点名称;`
   - 删除保存点：`RELEASE SAVEPOINT 保存点名称;`
   示例：
   ```
   START TRANSACTION;
   UPDATE user SET balance = balance - 100 WHERE id = 1;
   SAVEPOINT sp1; // 创建保存点sp1
   UPDATE user SET balance = balance + 100 WHERE id = 2;
   UPDATE user SET balance = balance + 50 WHERE id = 3; // 假设此操作失败
   ROLLBACK TO sp1; // 回滚到sp1，仅撤销id=3的修改，id=1的修改保留
   COMMIT; // 提交事务，最终仅id=1的修改生效
   ```

4. MySQL的事务隔离级别

事务隔离级别用于控制并发事务之间的干扰，MySQL默认隔离级别为`REPEATABLE READ`（可重复读），支持4种隔离级别（从低到高，隔离性越强，并发性能越弱）：

1. **READ UNCOMMITTED（读未提交）**：最低隔离级别，一个事务可以读取另一个事务未提交的修改。存在“脏读”问题（读取到未提交的无效数据），几乎不用。
2. **READ COMMITTED（读已提交）**：一个事务只能读取另一个事务已提交的修改。解决了“脏读”问题，但存在“不可重复读”问题（同一事务内多次查询同一数据，结果可能不同，因为其他事务已提交修改）。适用于多数互联网场景（如电商商品详情查询）。
3. **REPEATABLE READ（可重复读）**：MySQL默认级别，同一事务内多次查询同一数据，结果始终一致，不受其他事务提交的修改影响。解决了“不可重复读”问题，但存在“幻读”问题（同一事务内多次执行同一范围查询，结果行数可能不同，因为其他事务插入了新数据）。
4. **SERIALIZABLE（串行化）**：最高隔离级别，所有事务串行执行（同一时间只能执行一个事务），完全避免了并发问题，但并发性能极差，适用于数据一致性要求极高的场景（如金融核心交易）。

查看当前隔离级别：`SELECT @@transaction_isolation;`
修改隔离级别：`SET GLOBAL TRANSACTION ISOLATION LEVEL 隔离级别;`（需重启会话生效）

十四、什么是索引？MySQL中有哪些类型的索引？

1. 索引的核心定义

索引（Index）是MySQL中用于提升查询性能的一种数据结构（类似书籍的目录），它通过对表中的特定字段进行排序和存储，让MySQL无需遍历全表即可快速定位到目标数据，从而大幅提升查询效率。但索引会降低插入、修改、删除的性能（因为修改数据时需要同步维护索引），因此需合理创建索引。

2. 索引的核心作用

1. 提升查询速度：避免全表扫描，快速定位目标数据（如查询100万行数据中的某条记录，无索引可能需遍历100万行，有索引仅需几次查找）。
2. 保证数据唯一性：通过唯一索引（如主键索引）约束字段值的唯一性，避免重复数据。

3. MySQL中常见的索引类型

按“数据结构”“功能”“存储方式”可分为不同类型，核心常用类型如下：

（1）按数据结构分类

1. **B+树索引**：MySQL默认的索引数据结构，适用于范围查询、排序查询，所有数据都存储在叶子节点，叶子节点之间通过链表连接，查询效率稳定。InnoDB和MyISAM均支持B+树索引，是最常用的索引类型。
2. **哈希索引**：基于哈希表实现，适用于等值查询（=、!=），查询速度极快（O(1)时间复杂度），但不支持范围查询、排序查询。MySQL中Memory存储引擎默认使用哈希索引，InnoDB支持自适应哈希索引（自动为频繁查询的字段创建）。
3. **Full-Text索引（全文索引）**：专为文本搜索设计，支持对长文本字段（如VARCHAR、TEXT）进行关键词匹配查询，适用于博客、新闻等内容检索场景。MySQL 5.6及以上版本的InnoDB支持全文索引。
4. **R-Tree索引（空间索引）**：用于存储空间数据（如地理位置、地图坐标），支持空间范围查询，适用于GIS（地理信息系统）场景，使用较少。

（2）按功能分类

1. **主键索引（Primary Key）**：最核心的索引，用于唯一标识表中的每条记录，一张表只能有一个主键索引，字段值不能为空（NOT NULL）且唯一。创建表时通过 `PRIMARY KEY` 指定，InnoDB中主键索引是聚簇索引（索引与数据绑定）。
   示例：`CREATE TABLE user (id INT PRIMARY KEY AUTO_INCREMENT, ...);`
2. **唯一索引（Unique Index）**：保证字段值的唯一性，但允许字段值为NULL（一张表可多个唯一索引）。适用于需要唯一约束但字段可能为空的场景（如用户手机号、邮箱）。
   示例：`CREATE UNIQUE INDEX idx_user_phone ON user(phone);`
3. **普通索引（Normal Index）**：最基础的索引，仅用于提升查询速度，无唯一性约束，字段值可重复、可为空。适用于频繁查询的非唯一字段（如商品分类、用户年龄）。
   示例：`CREATE INDEX idx_user_age ON user(age);`
4. **复合索引（Composite Index）**：基于多个字段创建的索引，适用于多字段联合查询场景（如查询“年龄25且性别男”的用户）。复合索引遵循“最左前缀匹配原则”（查询时必须包含最左侧字段，否则索引失效）。
   示例：`CREATE INDEX idx_user_age_gender ON user(age, gender);`（支持age查询、age+gender查询，不支持单独gender查询）

（3）按存储方式分类（仅InnoDB）

1. **聚簇索引（Clustered Index）**：索引与数据存储在一起，索引的叶子节点就是数据本身，InnoDB中主键索引默认是聚簇索引。查询时找到索引即可直接获取数据，效率高。
2. **非聚簇索引（Secondary Index）**：索引与数据分离，索引的叶子节点存储的是主键值，查询时需先通过非聚簇索引找到主键，再通过主键索引获取数据（此过程称为“回表查询”）。普通索引、唯一索引、复合索引均为非聚簇索引。

十五、MySQL如何执行查询优化?

查询优化的核心目标是让MySQL避免全表扫描，尽量使用索引快速定位数据，减少查询过程中的资源消耗（CPU、内存、IO）。优化手段围绕“索引优化”“SQL语句优化”“表结构优化”“配置优化”展开，具体如下：

1. 索引优化（最核心的优化手段）

1. 为频繁查询的字段创建索引：优先为WHERE条件、JOIN关联字段、ORDER BY排序字段、GROUP BY分组字段创建索引。
2. 合理使用复合索引：多字段联合查询时，创建复合索引并遵循“最左前缀匹配原则”，避免重复创建单一字段索引（如已创建idx_age_gender，无需再创建idx_age）。
3. 避免索引失效：以下情况会导致索引失效，需规避：
   - 索引字段使用函数或运算：`SELECT * FROM user WHERE SUBSTR(username,1,3)='张三';`（应改为`WHERE username LIKE '张三%'`）
   - 索引字段使用!=、<>、NOT IN、IS NOT NULL：可能导致全表扫描，尽量用等值查询替代。
   - 模糊查询以%开头：`SELECT * FROM user WHERE username LIKE '%张三';`（索引失效，改为`'张三%'`）
   - 复合索引不满足最左前缀：如复合索引idx_age_gender，查询`WHERE gender='男'`（索引失效）。
   - 隐式类型转换：字段为VARCHAR类型，查询时用数字：`WHERE username=123;`（应改为`username='123'`）
4. 定期维护索引：删除冗余索引（重复、无用的索引）和碎片化索引，通过`EXPLAIN`分析索引使用情况。

2. SQL语句优化

1. 避免SELECT *：仅查询需要的字段，减少数据传输量和IO消耗，同时避免回表查询（非聚簇索引场景）。
   优化前：`SELECT * FROM user WHERE age=25;`
   优化后：`SELECT id, username FROM user WHERE age=25;`
2. 使用LIMIT限制查询结果：避免返回大量数据，尤其分页查询时必须加LIMIT。
   示例：`SELECT * FROM user ORDER BY create_time DESC LIMIT 10;`

3. 优化JOIN查询：
   - 优先使用INNER JOIN（内连接），避免LEFT JOIN/RIGHT JOIN（若无需左/右表全部数据）；
   - JOIN关联的表不宜过多（建议不超过3张），关联字段需创建索引；
   - 小表驱动大表：让小表作为驱动表（INNER JOIN会自动优化，LEFT JOIN需将小表放在左表）。
4. 优化子查询：将子查询改为JOIN查询，避免MySQL多次执行子查询（子查询效率低，JOIN更高效）。
   优化前：`SELECT * FROM user WHERE id IN (SELECT user_id FROM order WHERE status=1);`
   优化后：`SELECT u.* FROM user u INNER JOIN order o ON u.id = o.user_id WHERE o.status=1;`
5. 避免在WHERE子句中使用OR：OR会导致索引失效，可用UNION替代。
   优化前：`SELECT * FROM user WHERE age=25 OR username='张三';`
   优化后：`SELECT * FROM user WHERE age=25 UNION SELECT * FROM user WHERE username='张三';`（需确保两个查询字段一致）

3. 表结构优化

1. 选择合适的数据类型：遵循“最小够用”原则，减少存储空间和IO消耗：
   - 整数类型：优先用TINYINT（1字节）、SMALLINT（2字节）、INT（4字节），避免用BIGINT（8字节）；
   - 字符串类型：短文本用VARCHAR（可变长度，节省空间），长文本用TEXT（避免VARCHAR过长）；
   - 日期类型：用DATE（日期）、TIME（时间）、DATETIME（日期时间），避免用字符串存储日期（无法排序、查询效率低）。
2. 避免使用NULL值：NULL值会增加索引和查询的复杂度，可用默认值替代（如用0替代NULL的年龄，用空字符串替代NULL的备注）。
3. 分表分库：当单表数据量过大（超过1000万行）时，拆分表或库提升性能：
   - 水平分表：按数据范围（如时间）或哈希值拆分（如用户ID取模），每张表结构相同；
   - 垂直分表：将表中不常用的大字段（如TEXT）拆分到单独表，减少主表数据量；
   - 分库：按业务模块拆分数据库（如用户库、订单库），减轻单库压力。
4. 使用合适的存储引擎：核心业务表用InnoDB（支持事务、高并发），只读表用MyISAM（查询快）。

4. 配置优化（MySQL参数调整）

1. 调整缓存参数：
   - `innodb_buffer_pool_size`：InnoDB数据缓存池大小，建议设置为服务器内存的50%-70%（提升数据缓存命中率，减少磁盘IO）；
   - `query_cache_size`：查询缓存大小（MySQL 8.0已移除，5.7及以下版本可用，适用于只读场景）。

2. 调整并发参数：
   - `max_connections`：最大并发连接数，根据服务器配置调整（避免连接数不足导致无法连接）；
   - `innodb_thread_concurrency`：InnoDB并发线程数，建议设置为CPU核心数的2-4倍。
3. 调整日志参数：
   - `innodb_log_file_size`：InnoDB重做日志文件大小，建议设置为512MB-2GB（提升事务日志写入效率）；
   - `slow_query_log`：开启慢查询日志（设置为ON），记录执行时间超过`long_query_time`（默认10秒）的SQL，用于定位慢查询。

5. 工具辅助优化：EXPLAIN分析查询计划

EXPLAIN是MySQL的核心优化工具，通过`EXPLAIN + SQL语句`可查看SQL的执行计划（如是否使用索引、扫描行数、连接方式等），从而定位优化点。
示例：`EXPLAIN SELECT * FROM user WHERE age=25;`
关键字段说明：
- `type`：查询类型，取值从好到坏为：system > const > eq_ref > ref > range > index > ALL（ALL表示全表扫描，需优化）；
- `key`：实际使用的索引（NULL表示未使用索引）；
- `rows`：MySQL预估的扫描行数（行数越少，效率越高）；
- `Extra`：额外信息（如Using index表示使用覆盖索引，无需回表；Using filesort表示排序未使用索引，需优化）。

十六、解释SQL注入攻击以及如何防止？

1. SQL注入攻击的核心定义

SQL注入（SQL Injection）是一种常见的Web安全漏洞，攻击者通过在用户输入的参数中插入恶意SQL语句，欺骗数据库执行非预期的操作（如查询敏感数据、修改数据、删除表、甚至获取数据库权限）。其本质是开发者未对用户输入进行严格过滤，导致用户输入直接拼接到SQL语句中执行。

2. SQL注入攻击的示例

以用户登录功能为例，正常的SQL逻辑如下（假设用户输入用户名和密码，后端拼接到SQL中查询）：

Code block

```
// 后端代码（未过滤用户输入，存在注入漏洞）
const username = req.body.username;
const password = req.body.password;
const sql = `SELECT * FROM user WHERE username='${username}' AND
password='${password}'`;
// 执行SQL查询，若有结果则登录成功
```

攻击者可通过以下输入实施注入：

1. 用户名输入：`' OR '1'='1`，密码任意输入，拼接后的SQL为：
   `SELECT * FROM user WHERE username='' OR '1'='1' AND password='任意值'`
   由于`'1'='1`恒成立，此SQL会查询出表中所有用户数据，攻击者无需正确密码即可登录。
2. 用户名输入：`'; DROP TABLE user; --`，拼接后的SQL为：
   `SELECT * FROM user WHERE username=''; DROP TABLE user; --' AND password='任意值'`
   其中`;`用于结束前一个SQL语句，`DROP TABLE user;`删除user表，`--`注释掉后续内容，导致数据库表被恶意删除。

3. SQL注入攻击的危害

1. 获取敏感数据：如用户密码、手机号、银行卡号等核心信息；
2. 篡改数据：修改用户余额、订单状态等业务数据；
3. 破坏数据：删除表、清空数据，导致业务瘫痪；
4. 获取数据库权限：进一步控制服务器，实施更严重的攻击（如植入木马）。

4. 防止SQL注入的核心措施

防止SQL注入的核心原则是“不相信任何用户输入”，通过严格过滤、参数化查询等方式，避免用户输入直接拼接到SQL语句中。具体措施如下：

1. **使用参数化查询（最有效，推荐）**：将用户输入作为参数传递给SQL语句，而非直接拼接，数据库会自动过滤恶意输入，避免SQL注入。不同开发语言/框架的实现方式不同，Node.js中常用的mysql模块支持`?`占位符实现参数化查询。
   示例（Node.js mysql模块参数化查询）：
   ```
   const username = req.body.username;
   const password = req.body.password;
   // 用?作为占位符，参数放在数组中传递
   const sql = 'SELECT * FROM user WHERE username=? AND password=?';
   connection.query(sql, [username, password], (err, result) => {
     // 处理查询结果
   });
   ```
   说明：此时用户输入的恶意字符（如'、;、--）会被当作普通字符串处理，不会被解析为SQL语句的一部分。

2. **使用ORM框架（对象关系映射）**：ORM框架（如Node.js的Sequelize、Java的MyBatis-Plus）内部已实现参数化查询，开发者无需手动拼接SQL，从根本上避免注入风险。
   示例（Sequelize查询）：
   ```
   // 定义User模型
   const User = sequelize.define('user', {
     username: DataTypes.STRING,
     password: DataTypes.STRING
   });
   // 查询用户（ORM自动处理参数，无注入风险）
   const user = await User.findOne({
     where: { username: req.body.username, password: req.body.password }
   });
   ```
3. **严格过滤用户输入**：对用户输入的特殊字符（如'、"、;、--、OR、AND）进行转义或过滤，尤其适用于无法使用参数化查询的场景（需注意：过滤无法覆盖所有注入场景，仅作为辅助手段）。
   示例（转义单引号）：
   ```
   const username = req.body.username.replace(/'/g, '\\''); // 将'转义为\'，避免闭合SQL字符串
   ```
4. **限制数据库用户权限**：遵循“最小权限原则”，为应用程序使用的数据库用户分配最小必要权限（如仅授予SELECT、INSERT、UPDATE权限，禁止授予DROP、ALTER、CREATE等高危权限），即使发生注入攻击，攻击者也无法执行破坏操作。
5. **避免暴露敏感信息**：
   - 关闭数据库错误信息的详细输出（如MySQL的错误堆栈信息），避免攻击者通过错误信息推测数据库结构；
   - 不向用户返回数据库相关的错误信息，统一返回通用错误提示（如“登录失败，请重试”）。
6. **使用 PreparedStatement（预处理语句）**：数据库层面的预处理机制，将SQL语句模板和参数分开传输，数据库先编译SQL模板，再代入参数执行，避免参数被解析为SQL代码。参数化查询本质就是基于PreparedStatement实现的。
7. **定期更新数据库和框架版本**：及时修复数据库和开发框架中的安全漏洞，减少注入攻击的可乘之机。

十七、什么是服务端渲染(SSR)？它如何提高前端应用的首屏加载时间？

1. 服务端渲染（SSR）的核心定义

服务端渲染（Server-Side Rendering，简称SSR）是一种前端应用的渲染方式，指**页面的HTML结构在服务器端生成，服务器将完整的、包含数据的HTML页面发送给浏览器，浏览器直接解析HTML即可展示页面**。与之相对的是客户端渲染（CSR，Client-Side Rendering）：浏览器先获取空白HTML和JavaScript文件，再通过JS请求数据、生成DOM并渲染页面。

常见的SSR框架：React生态的Next.js、Vue生态的Nuxt.js、Node.js原生+模板引擎（如EJS、Pug）。

2. SSR的核心工作流程

1. 用户发送HTTP请求到服务器（如访问https://www.example.com）；
2. 服务器接收请求后，执行后端代码（Node.js等），获取页面所需的数据（如从数据库查询用户信息、商品数据）；
3. 服务器将获取的数据注入到HTML模板中，生成完整的、包含数据的HTML页面（HTML中已包含页面的所有内容，无需浏览器再请求数据生成）；
4. 服务器将完整的HTML页面发送给浏览器；
5. 浏览器接收HTML后，直接解析HTML、CSS，快速展示页面（此时页面已可交互或部分可交互）；
6. 浏览器异步加载页面所需的JavaScript文件，执行JS代码（如绑定事件、初始化交互功能），完成页面的“ hydration（水合）”过程，最终形成完全可交互的页面。

3. SSR如何提高前端应用的首屏加载时间?

首屏加载时间是指用户输入URL后，直到首屏内容完全展示的时间。SSR通过优化“首屏内容生成和传输”的流程，大幅缩短首屏加载时间，核心原因如下：

1. **减少浏览器的请求次数和等待时间**：
   - 客户端渲染（CSR）需要3次核心请求：①获取空白HTML → ②获取JS/CSS资源 → ③请求数据（通过JS），只有完成这3次请求才能生成并渲染页面，等待时间长；
   - SSR只需1次核心请求：获取完整的HTML页面（已包含数据和CSS），浏览器无需再请求数据，减少了请求次数和网络等待时间，首屏内容可快速展示。

1. **HTML内容直达，无需浏览器等待JS执行**：
   - CSR中，空白HTML本身无内容，必须等待JS加载完成并执行、请求到数据后，才能生成DOM并渲染页面；若JS文件较大（如React、Vue框架代码），加载和执行时间会很长，首屏会出现“白屏”；
   - SSR中，服务器直接发送包含完整内容的HTML，浏览器接收后可立即解析HTML和CSS，展示首屏内容，无需等待JS加载和执行（JS加载和水合过程可在首屏展示后异步进行），彻底解决首屏白屏问题。

1. **利用服务器的网络优势，减少数据请求延迟**：
   - CSR中，数据请求由浏览器发起，浏览器与数据库之间可能存在跨域、网络链路长等问题，数据请求延迟高；
   - SSR中，数据请求由服务器发起，服务器通常与数据库部署在同一机房（或内网），网络链路短，数据获取速度远快于浏览器，能快速生成包含数据的HTML，提升首屏加载效率。

1. **优化资源加载优先级：**
   - SSR中，HTML和关键CSS直接嵌入页面或优先加载，非关键JS异步加载，确保首屏内容的渲染资源优先获取；
   - CSR中，JS是核心渲染依赖，必须等待JS加载完成才能渲染，资源加载优先级不合理，首屏加载慢。

4. SSR的优缺点补充

（1）优点：① 首屏加载快，提升用户体验；② 有利于SEO（搜索引擎可直接抓取完整的HTML内容，CSR的空白HTML难以被抓取）；③ 兼容性好，支持不支持JS的浏览器。

（2）缺点：① 服务器压力大（需要生成HTML、处理数据请求）；② 开发复杂度高（需处理服务端与客户端的状态同步、水合等问题）；③ 部署成本高（需要Node.js等服务端运行环境）。