实现一个简化版的 Promise 类，核心是理解 Promise 的状态管理、执行流程、异步行为和链式调用。

基础功能要求

1. 构造函数（constructor）

- 接收一个执行器函数 executor，立即同步执行。
- executor 接收两个回调参数：resolve 和 reject。
- Promise 有三种状态：
    - pending（等待中，初始状态）
    - fulfilled（成功）
    - rejected（失败）
- 调用 resolve(value) 将状态从 pending 变为 fulfilled，保存 value。
- 调用 reject(reason) 将状态从 pending 变为 rejected，保存 reason。
- 如果执行器抛错，自动调用 reject 并传入错误。

2. then 方法

- 接收两个回调参数：onFulfilled 和 onRejected（均可选）。
- 根据当前 Promise 状态，调用相应的回调。
- 支持多个 .then 被调用时，回调以队列形式存储。
- 如果状态是 pending，先存储回调，待状态变更后调用。
- .then 返回一个新的 Promise，支持链式调用。

Follow-up 扩展要求（更贴近原生 Promise 行为）

3. 异步回调执行

- .then 中的回调函数必须异步执行（使用 setTimeout 或微任务，如 queueMicrotask / process.nextTick / Promise.resolve().then()）。

- 确保在调用 resolve 之前或之后调用 .then，回调都能被正确注册并异步触发。

4. 支持链式调用

- .then 返回一个新的 Promise，支持链式调用。
- 链式中，.then 的回调返回值：
    - 如果是普通值，新的 Promise 以该值 resolve。
    - 如果是 Promise，则新的 Promise 采用该 Promise 的状态和结果（递归解析）。
- 需要实现辅助函数（通常称为 resolvePromise）来处理 then 返回的 Promise，避免循环引用和错误。

5. 处理多个 .then 的回调队列

- 支持在同一个 Promise 上调用多个 .then，它们的回调都要被依次调用。

额外（可能被问答的）内容

6. 实现 Promise.all 静态方法

- 接收一个 Promise 数组，返回一个新的 Promise。
- 只有当所有 Promise 都 fulfilled，才 fulfilled 并返回结果数组。
- 只要有一个 Promise rejected，就立即 rejected 并返回对应原因。

面试时的思路建议

1. 先实现**同步**执行的基本版本（executor 立即调用，then 立即执行对应回调）。
2. 再改进支持**异步**执行（用 setTimeout 或微任务异步调用回调）。
3. 然后支持在 resolve 之前和之后调用 .then，确保回调都能执行。
4. 实现链式 .then，返回新的 Promise，并正确处理链式返回值。
5. 如果时间允许，实现 Promise.all。

总结示例的关键点

功能	要点	细节
构造函数	执行 executor，状态管理，异常处理	状态变更且只变一次
then 方法	接受成功/失败回调，状态判断调用	多次调用支持，回调队列
异步回调	异步调用回调函数	使用微任务或 setTimeout
链式调用	then 返回新 Promise	支持返回 Promise 或普通值
处理返回值	递归处理 then 返回的 Promise	避免循环引用
Promise.all	静态方法，等待所有 Promise 完成	有一个失败就失败

```
class MyPromise {
  constructor(executor) {
    this.state = "pending"; // Promise 状态： pending, fulfilled, rejected
    this.value = undefined; // fulfilled 时的值
    this.reason = undefined; // rejected 时的原因

    this.onFulfilledCallbacks = []; // fulfilled 状态回调队列
    this.onRejectedCallbacks = []; // rejected 状态回调队列

    const resolve = (value) => {
      if (this.state !== "pending") return;

      // 如果 resolve 的值是一个 Promise，则等待其完成
```

```
  if (value instanceof MyPromise) {
    return value.then(resolve, reject);
  }

  this.state = "fulfilled";
  this.value = value;

  // 微任务执行所有成功回调
  Promise.resolve().then(() => {
    this.onFulfilledCallbacks.forEach((fn) => fn(this.value));
  });
};

const reject = (reason) => {
  if (this.state !== "pending") return;

  this.state = "rejected";
  this.reason = reason;

  // 微任务执行所有失败回调
  Promise.resolve().then(() => {
    this.onRejectedCallbacks.forEach((fn) => fn(this.reason));
  });
};

// 立即执行 executor，捕获异常自动 reject
```

```
  try {
    executor(resolve, reject);
  } catch (error) {
    reject(error);
  }
}

then(onFulfilled, onRejected) {
  // 如果不是函数，提供默认函数传递值或抛错
  onFulfilled = typeof onFulfilled === "function" ? onFulfilled : (v) => v;
  onRejected =
    typeof onRejected === "function"
      ? onRejected
      : (err) => {
          throw err;
        };

  // then 返回一个新的 Promise，支持链式调用
  const promise2 = new MyPromise((resolve, reject) => {
    if (this.state === "fulfilled") {
      Promise.resolve().then(() => {
        try {
          const x = onFulfilled(this.value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
```

```
    }
  });
} else if (this.state === "rejected") {
  Promise.resolve().then(() => {
    try {
      const x = onRejected(this.reason);
      resolvePromise(promise2, x, resolve, reject);
    } catch (e) {
      reject(e);
    }
  });
} else if (this.state === "pending") {
  // 状态为 pending 时，将回调保存，状态改变后再执行
  this.onFulfilledCallbacks.push(() => {
    Promise.resolve().then(() => {
      try {
        const x = onFulfilled(this.value);
        resolvePromise(promise2, x, resolve, reject);
      } catch (e) {
        reject(e);
      }
    });
  });

  this.onRejectedCallbacks.push(() => {
    Promise.resolve().then(() => {
```

```
        try {
          const x = onRejected(this.reason);
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          reject(e);
        }
       });
     });
   }
 });

 return promise2;
}

// 实现静态方法 Promise.all
static all(promises) {
 return new MyPromise((resolve, reject) => {
   if (!Array.isArray(promises)) {
     return reject(new TypeError("Argument must be an array"));
   }

   const results = [];
   let completedCount = 0;

   if (promises.length === 0) {
     resolve(results);
```

```
      return;
    }

    promises.forEach((p, index) => {
      // 兼容非 Promise 值
      MyPromise.resolve(p).then(
        (value) => {
          results[index] = value;
          completedCount++;
          if (completedCount === promises.length) {
            resolve(results);
          }
        },
        (err) => {
          reject(err);
        }
      );
    });
  });
}

// 静态 resolve 方法，兼容普通值和 Promise
static resolve(value) {
  if (value instanceof MyPromise) {
    return value;
  }
```

```
    return new MyPromise((resolve) => resolve(value));
  }
}

// 辅助函数：处理 then 返回值，兼容 Promise/A+ 规范
function resolvePromise(promise2, x, resolve, reject) {
  if (promise2 === x) {
    return reject(new TypeError("Chaining cycle detected for promise"));
  }

  if (x !== null && (typeof x === "object" || typeof x === "function")) {
    let called = false;
    try {
      let then = x.then;
      if (typeof then === "function") {
        then.call(
          x,
          (y) => {
            if (called) return;
            called = true;
            resolvePromise(promise2, y, resolve, reject);
          },
          (r) => {
            if (called) return;
            called = true;
            reject(r);
```

```
        }
      );
    } else {
      if (called) return;
      called = true;
      resolve(x);
    }
  } catch (e) {
    if (called) return;
    called = true;
    reject(e);
  }
 } else {
  resolve(x);
 }
}

// ===== 测试用例 =====

function testSyncResolve() {
 const p = new MyPromise((resolve) => {
  resolve(42);
 });

 p.then((value) => {
  console.log("Sync resolve:", value); // Sync resolve: 42
```

```
  });
}

function testAsyncResolve() {
  const p = new MyPromise((resolve) => {
    setTimeout(() => {
      resolve("async value");
    }, 100);
  });

  p.then((value) => {
    console.log("Async resolve:", value); // Async resolve: async value
  });
}

function testThenBeforeResolve() {
  const p = new MyPromise((resolve) => {
    setTimeout(() => {
      resolve("late resolve");
    }, 100);
  });

  p.then((value) => {
    console.log("Then before resolve:", value); // Then before resolve: late resolve
  });
}
```

```
function testChain() {
  new MyPromise((resolve) => {
    resolve(10);
  })
    .then((x) => {
      console.log("Chain step 1:", x); // Chain step 1: 10
      return x * 2;
    })
    .then((x) => {
      console.log("Chain step 2:", x); // Chain step 2: 20
      return new MyPromise((resolve) => {
        setTimeout(() => resolve(x + 5), 50);
      });
    })
    .then((x) => {
      console.log("Chain step 3 async:", x); // Chain step 3 async: 25
    });
}

function testReject() {
  new MyPromise((_, reject) => {
    reject("error happened");
  }).then(null, (err) => {
    console.log("Rejected:", err); // Rejected: error happened
  });
```

```
}

function testPromiseAll() {
  const p1 = new MyPromise((resolve) => setTimeout(() => resolve(1), 100));
  const p2 = new MyPromise((resolve) => setTimeout(() => resolve(2), 50));
  const p3 = 3; // 普通值

  MyPromise.all([p1, p2, p3])
    .then((results) => {
      console.log("Promise.all results:", results); // Promise.all results: [1, 2, 3]
    })
    .catch((err) => {
      console.error("Promise.all error:", err);
    });
}

// 运行所有测试
testSyncResolve();
testAsyncResolve();
testThenBeforeResolve();
testChain();
testReject();
testPromiseAll();
```