Promise

`JavaScript` 中存在很多异步操作,`Promise` 将异步操作队列化，按照期望的顺序执行，返回符合预期的结果。可以通过链式调用多个 `Promise` 达到我们的目的。
Promise 在各种开源库中已经实现，现在标准化后被浏览器默认支持。

> promise 是一个拥有 `then` 方法的对象或函数

#问题探讨

下面通过多个示例来感受一下不使用 `promise` 时，处理相应问题的不易，及生成了不便阅读的代码。

#定时嵌套

下面是一个定时器执行结束后，执行另一个定时器，这种嵌套造成代码不易阅读

```
<style>
  div {
    width: 100px;
    height: 100px;
    background: yellowgreen;
    position: absolute;
  }
</style>

<body>
  <div></div>
</body>

<script>
  function interval(callback, delay = 100) {
    let id = setInterval(() => callback(id), delay);
  }

  const div = document.querySelector("div");
  interval(timeId => {
    const left = parseInt(window.getComputedStyle(div).left);
    div.style.left = left + 10 + "px";
    if (left > 200) {
```

```
      clearInterval(timeId);
      interval(timeId => {
        const width = parseInt(window.getComputedStyle(div).width);
        div.style.width = width - 1 + "px";
        if (width <= 0) clearInterval(timeId);
      }, 10);
    }
  }, 100);
</script>

```

#图片加载

下面是图片后设置图片边框，也需要使用回调函数处理，代码嵌套较复杂

```
function loadImage(file, resolve, reject) {
  const image = new Image();
  image.src = file;
  image.onload = () => {
    resolve(image);
  };
  image.onerror = () => {
    reject(new Error("load fail"));
  };
  document.body.appendChild(image);
}

loadImage(
  "images/houdunren.png",
  image => {
    image.style.border = "solid 5px red";
  },
  error => {
    console.log(error);
  }
);

```

#加载文件

下面是异步加载外部JS文件，需要使用回调函数执行，并设置的错误处理的回调函数

```
function load(file, resolve, reject) {
  const script = document.createElement("script");
  script.src = file;
  script.onload = resolve;
  script.onerror = reject;
  document.body.appendChild(script);
}
load(
  "js/hd.js",
  script => {
    console.log(`${script.path[0].src} 加载成功`);
    hd();
  },
  error => {
    console.log(`${error.srcElement.src} 加载失败`);
  }
);

```

实例中用到的 `hd.js` 与 `houdunren.js` 内容如下

```
# hd.js
function hd() {
  console.log("hd function run");
}

# houdunren.js
function houdunren() {
  console.log("houdunren function run");
  hd();
}

```

如果要加载多个脚本时需要嵌套使用，下面`houdunren.js` 依赖 `hd.js`，需要先加载`hd.js` 后加载 `houdunren.js`

> 不断的回调函数操作将产生回调地狱，使代码很难维护

```
load(
  "js/hd.js",
  script => {
    load(
      "js/houdunren.js",
      script => {
        houdunren();
      },
      error => {
        console.log(`${error.srcElement.src} 加载失败`);
      }
    );
  },
  error => {
    console.log(`${error.srcElement.src} 加载失败`);
  }
);

```

#异步请求

使用传统的异步请求也会产生回调嵌套的问题，下在是获取向军的成绩，需要经过以下两步

1. 根据用户名取得

向军 的编号

2. 根据编号获取成绩

> 示例中用到的 php 文件请在 版本库 (opens new window)中查看

启动 PHP 服务器命令 `php -S localhost:8080`

```
function ajax(url, resolve, reject) {
  let xhr = new XMLHttpRequest();
  xhr.open("GET", url);
  xhr.send();
  xhr.onload = function() {
    if (this.status == 200) {
      resolve(JSON.parse(this.response));
    } else {
      reject(this);
```

```
    }
  };
}
ajax("http://localhost:8888/php/user.php?name=
ajax("
  ajax(
    `http://localhost:8888/php/houdunren.php?id=
    `
    response => {
      console.log(response[0]);
    }
  );
});

```

#肯德基

下面是模拟肯德基吃饭的事情，使用 `promise` 操作异步的方式每个阶段会很清楚

```
let kfc = new Promise((resolve, reject) => {
  console.log("肯德基厨房开始做饭");
  resolve("我是肯德基，你的餐已经做好了");
});
let dad = kfc.then(msg => {
  console.log(`收到肯德基消息: ${msg}`);
  return {
    then(resolve) {
      setTimeout(() => {
        resolve("孩子，我吃了两秒了，不辣，你可以吃了");
      }, 2000);
    }
  };
});
let son = dad.then(msg => {
  return new Promise((resolve, reject) => {
    console.log(`收到爸爸消息: ${msg}`);
    setTimeout(() => {
      resolve("妈妈，我和向军爸爸吃完饭了");
```

```
    }, 2000);
  });
});
let ma = son.then(msg => {
  console.log(`收到孩子消息: ${msg},事情结束`);
});

```

而使用以往的回调方式，就会让人苦不堪言

```
function notice(msg, then) {
  then(msg);
}
function meal() {
  notice("肯德基厨房开始做饭", msg => {
    console.log(msg);
    notice("我是肯德基，你的餐已经做好", msg => {
      console.log(`收到肯德基消息: ${msg}`);
      setTimeout(() => {
        notice("孩子，我吃了两秒了，不辣，你可以吃了", msg => {
          console.log(`收到爸爸消息: ${msg}`);
          setTimeout(() => {
            notice("妈妈，我和向军爸爸吃完饭了", msg => {
              console.log(`收到孩子消息: ${msg},事情结束`);
            });
          }, 2000);
        });
      }, 2000);
    });
  });
}
meal();

```

#异步状态

Promise 可以理解为承诺，就像我们去 KFC 点餐服务员给我们一引取餐票，这就是承诺。如果餐做好了叫我们这就是成功，如果没有办法给我们做出食物这就是拒绝。

- 一个

promise 必须有一个
then 方法用于处理状态改变

#状态说明

Promise 包含`pending`、`fulfilled`、`rejected`三种状态

- pending 指初始等待状态，初始化

promise 时的状态

- resolve 指已经解决，将

promise 状态设置为
fulfilled

- reject 指拒绝处理，将

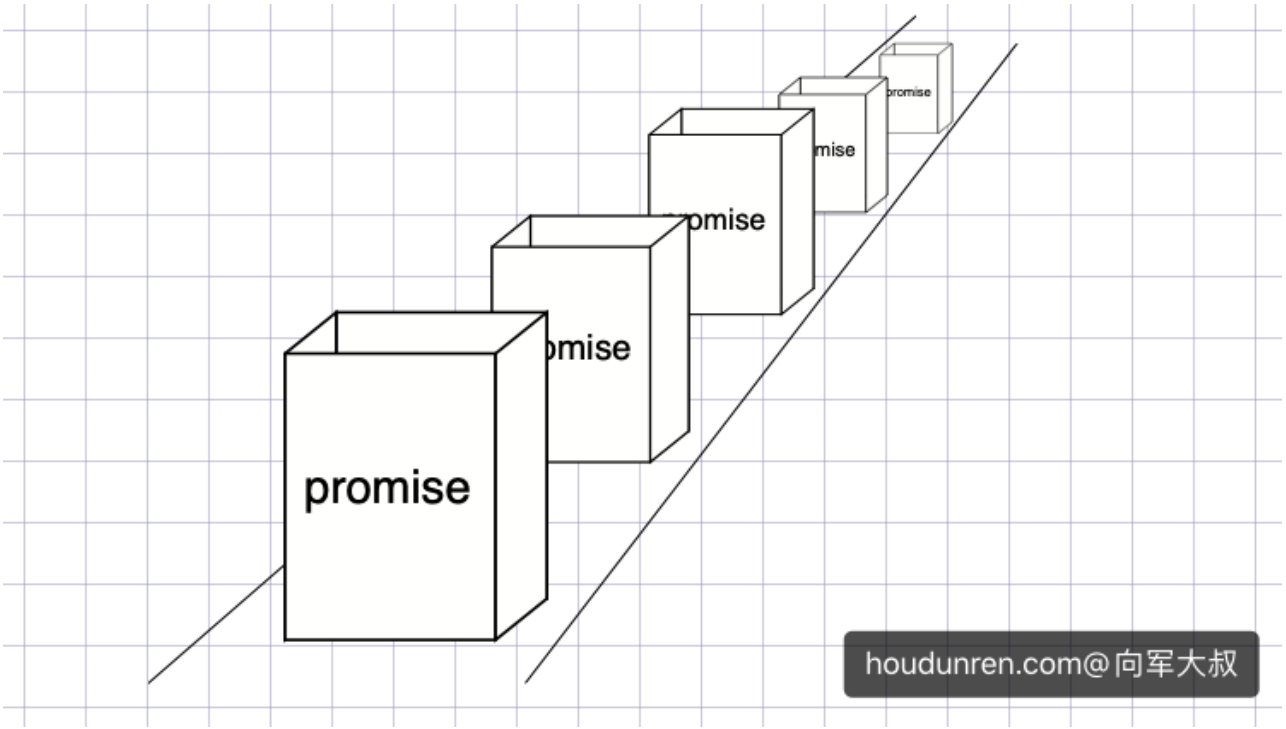
promise 状态设置为
rejected

- promise 是生产者，通过

resolve 与
reject 函数告之结果

- promise 非常适合需要一定执行时间的异步任务
- 状态一旦改变将不可更改

promise 是队列状态，就像体育中的接力赛，或多米诺骨牌游戏，状态一直向后传递，当然其中的任何一个 promise 也可以改变状态。



promise 没有使用 `resolve` 或 `reject` 更改状态时，状态为 `pending`

```
console.log(
  new Promise((resolve, reject) => {
  });
); //Promise {<pending>}

```

当更改状态后

```
console.log(
  new Promise((resolve, reject) => {
    resolve("fulfilled");
  })
); //Promise {<resolved>: "fulfilled"}

console.log(
  new Promise((resolve, reject) => {
    reject("rejected");
  })
); //Promise {<rejected>: "rejected"}

```

`promise` 创建时即立即执行即同步任务，`then` 会放在异步微任务中执行，需要等同步任务执行后才执行。

```
let promise = new Promise((resolve, reject) => {
  resolve("fulfilled");
  console.log("后盾人");
});
promise.then(msg => {
  console.log(msg);
});
console.log("houdunren.com");

```

`promise` 操作都是在其他代码后执行，下面会先输出 `houdunren.com` 再弹出 `success`

- promise 的 then、catch、finally 的方法都是异步任务
- 程序需要将主任务执行完成才会执行异步队列任务

```
const promise = new Promise(resolve => resolve("success"));
promise.then(alert);
alert("houdunren.com");
promise.then(() => {
  alert("后盾人");
});

```

下例在三秒后将 `Promise` 状态设置为 `fulfilled` ， 然后执行 `then` 方法

```
new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("fulfilled");
  }, 3000);
}).then(
  msg => {
    console.log(msg);
  },
  error => {
    console.log(error);
  }
);

```

状态被改变后就不能再修改了，下面先通过`resolve` 改变为成功状态，表示`promise` 状态已经完成，就不能使用 `reject` 更改状态了

```
new Promise((resolve, reject) => {
  resolve("操作成功");
  reject(new Error("请求失败"));
}).then(
  msg => {
    console.log(msg);
  },
  error => {
    console.log(error);
  }
```

```
);

```

#动态改变

下例中 p2 返回了 p1 所以此时 p2 的状态已经无意义了，后面的 then 是对 p1 状态的处理。

```
const p1 = new Promise((resolve, reject) => {
  // resolve("fulfilled");
  reject("rejected");
});
const p2 = new Promise(resolve => {
  resolve(p1);
}).then(
  value => {
    console.log(value);
  },
  reason => {
    console.log(reason);
  }
);

```

如果 `resolve` 参数是一个 `promise` ，将会改变`promise`状态。

下例中 `p1` 的状态将被改变为 `p2` 的状态

```
const p1 = new Promise((resolve, reject) => {
  resolve(
        //p2
    new Promise((s, e) => {
      s("成功");
    })
  );
}).then(msg => {
  console.log(msg);
});

```

当 promise 做为参数传递时，需要等待 promise 执行完才可以继承，下面的 p2 需要等待 p1 执行完成。

- 因为

p2 的

resolve 返回了

p1 的 promise，所以此时

p2 的

then 方法已经是

p1 的了

- 正因为以上原因

then 的第一个函数输出了

p1 的

resolve 的参数

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("操作成功");
  }, 2000);
});
const p2 = new Promise((resolve, reject) => {
  resolve(p1);
}).then(
  msg => {
    console.log(msg);
  },
  error => {
    console.log(error);
  }
);

```

#then

一个 promise 需要提供一个 then 方法访问 promise 结果，`then` 用于定义当 `promise` 状态发生改变时的处理，即`promise`处理异步操作，`then` 用于结果。

`promise` 就像 `kfc` 中的厨房，`then` 就是我们用户，如果餐做好了即 `fulfilled` ，做不了拒绝即 `rejected` 状态。那么 then 就要对不同状态处理。

- then 方法必须返回 promise，用户返回或系统自动返回

- 第一个函数在

resolved 状态时执行，即执行

resolve时执行

then第一个函数处理成功状态

- 第二个函数在

rejected状态时执行，即执行

reject 时执行第二个函数处理失败状态，该函数是可选的

- 两个函数都接收

promise 传出的值做为参数

- 也可以使用

catch 来处理失败的状态

- 如果

then 返回

promise ， 下一个

then 会在当前

promise 状态改变后执行

#语法说明

then 的语法如下，onFulfilled 函数处理 `fulfilled` 状态， onRejected 函数处理 `rejected` 状态

- onFulfilled 或 onRejected 不是函数将被忽略
- 两个函数只会被调用一次
- onFulfilled 在 promise 执行成功时调用
- onRejected 在 promise 执行拒绝时调用

```
promise.then(onFulfilled, onRejected)

```

#基础知识

`then` 会在 `promise` 执行完成后执行， `then` 第一个函数在 `resolve`成功状态执行

```
const promise = new Promise((resolve, reject) => {
  resolve("success");
}).then(
  value => {
    console.log(`解决: ${value}`);
  },
```

```
  reason => {
    console.log(`拒绝:${reason}`);
  }
);

```

`then` 中第二个参数在失败状态执行

```
const promise = new Promise((resolve, reject) => {
  reject("is error");
});
promise.then(
  msg => {
    console.log(`成功: ${msg}`);
  },
  error => {
    console.log(`失败:${error}`);
  }
);

```

如果只关心成功则不需要传递 `then` 的第二个参数

```
const promise = new Promise((resolve, reject) => {
  resolve("success");
});
promise.then(msg => {
  console.log(`成功: ${msg}`);
});

```

如果只关心失败时状态，`then` 的第一个参数传递 `null`

```
const promise = new Promise((resolve, reject) => {
  reject("is error");
});
promise.then(null, error => {
  console.log(`失败:${error}`);
});
```

```
7
```

promise 传向 then 的传递值，如果 then 没有可处理函数，会一直向后传递

```
let p1 = new Promise((resolve, reject) => {
 reject("rejected");
})
.then()
.then(
  null,
  f => console.log(f)
);

```

如果 onFulfilled 不是函数且 promise 执行成功, p2 执行成功并返回相同值

```
let promise = new Promise((resolve, reject) => {
  resolve("resolve");
});
let p2 = promise.then();
p2.then().then(resolve => {
  console.log(resolve);
});

```

如果 onRejected 不是函数且 promise 拒绝执行，p2 拒绝执行并返回相同值

```
let promise = new Promise((resolve, reject) => {
  reject("reject");
});
let p2 = promise.then(() => {});
p2.then(null, null).then(null, reject => {
  console.log(reject);
});

```

#链式调用

每次的 `then` 都是一个全新的 `promise`，默认 then 返回的 promise 状态是 fulfilled

```
let promise = new Promise((resolve, reject) => {
  resolve("fulfilled");
}).then(resolve => {
  console.log(resolve);
})
.then(resolve => {
  console.log(resolve);
});

```

每次的 `then` 都是一个全新的 `promise`，不要认为上一个 promise 状态会影响以后 then 返回的状态

```
let p1 = new Promise(resolve => {
  resolve();
});
let p2 = p1.then(() => {
  console.log("后盾人");
});
p2.then(() => {
  console.log("houdunren.com");
});
console.log(p1); // Promise {<resolved>}
console.log(p2); // Promise {<pending>}

# 再试试把上面两行放在 setTimeout里
setTimeout(() => {
  console.log(p1); // Promise {<resolved>}
  console.log(p2); // Promise {<resolved>}
});

```

`then` 是对上个 promise 的`rejected` 的处理，每个 `then` 会是一个新的 promise，默认传递 `fulfilled` 状态

```
new Promise((resolve, reject) => {
  reject();
})
.then(
```

```
  resolve => console.log("fulfilled"),
  reject => console.log("rejected")
)
.then(
  resolve => console.log("fulfilled"),
  reject => console.log("rejected")
)
.then(
  resolve => console.log("fulfilled"),
  reject => console.log("rejected")
);

# 执行结果如下
  ejected
  fulfilled
  fulfilled

```

如果内部返回 `promise` 时将使用该 `promise`

```
let p1 = new Promise(resolve => {
  resolve();
});
let p2 = p1.then(() => {
  return new Promise(r => {
    r("houdunren.com");
  });
});
p2.then(v => {
  console.log(v); //houdunren.com
});

```

如果 `then` 返回`promise` 时，后面的`then` 就是对返回的 `promise` 的处理，需要等待该 promise 变更状态后执行。

```
let promise = new Promise(resolve => resolve());
let p1 = promise.then(() => {
  return new Promise(resolve => {
```

```
    setTimeout(() => {
      console.log(`p1`);
      resolve();
    }, 2000);
  });
}).then(() => {
  return new Promise((a, b) => {
    console.log(`p2`);
  });
});

```

如果then返回 promise 时，返回的promise 后面的then 就是处理这个promise 的

> 如果不 return 情况就不是这样了，即外层的 then 的promise 和内部的promise 是独立的两个 promise

```
new Promise((resolve, reject) => {
  resolve();
})
.then(v => {
  return new Promise((resolve, reject) => {
    resolve("第二个promise");
  }).then(value => {
    console.log(value);
    return value;
  });
})
.then(value => {
  console.log(value);
});

```

这是对上面代码的优化，把内部的 then 提取出来

```
new Promise((resolve, reject) => {
  resolve();
})
.then(v => {
  return new Promise((resolve, reject) => {
```

```
    resolve("第二个promise");
  });
})
.then(value => {
  console.log(value);
  return value;
})
.then(value => {
  console.log(value);
});

```

#其它类型

Promise 解决过程是一个抽象的操作，其需输入一个 `promise` 和一个值，我们表示为 `[[Resolve]](promise, x)`，如果 `x` 有 `then` 方法且看上去像一个 Promise ，解决程序即尝试使 `promise` 接受 `x` 的状态；否则其用 `x` 的值来执行 `promise` 。

#循环调用

如果 `then` 返回与 `promise` 相同将禁止执行

```
let promise = new Promise(resolve => {
  resolve();
});
let p2 = promise.then(() => {
  return p2;
}); // TypeError: Chaining cycle detected for promise

```

#promise

如果返加值是 `promise` 对象，则需要更新状态后，才可以继承执行后面的`promise`

```
new Promise((resolve, reject) => {
  resolve(
    new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve("解决状态");
```

```
      }, 2000);
    })
  );
})
  .then(
    v => {
      console.log(`fulfilled: ${v}`);
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          reject("失败状态");
        }, 2000);
      });
    },
    v => {
      console.log(`rejected: ${v}`);
    }
  )
  .catch(error => console.log(`rejected: ${error}`));

```

#Thenables

包含 `then` 方法的对象就是一个 `promise` ，系统将传递 resolvePromise 与 rejectPromise 做为函数参数

下例中使用 `resolve` 或在`then` 方法中返回了具有 `then`方法的对象

- 该对象即为

promise 要先执行，并在方法内部更改状态

- 如果不更改状态，后面的

then promise 都为等待状态

```
new Promise((resolve, reject) => {
  resolve({
    then(resolve, reject) {
      resolve("解决状态");
    }
  });
})
.then(v => {
  console.log(`fulfilled: ${v}`);
```

```
  return {
    then(resolve, reject) {
      setTimeout(() => {
        reject("失败状态");
      }, 2000);
    }
  };
})
.then(null, error => {
  console.log(`rejected: ${error}`);
});

```

包含 `then` 方法的对象可以当作 promise 来使用

```
class User {
  (id) {
    this.id = id;
  }
  then(resolve, reject) {
    resolve(ajax(`
http://localhost:8888/php/houdunren.php?id=
    resolve(ajax(`
  }
}
new Promise((resolve, reject) => {
  resolve(ajax(`
http://localhost:8888/php/user.php?name=
  resolve(ajax(`
})
.then(user => {
  return new User(user.id);
})
.then(lessons => {
  console.log(lessons);
});

```

当然也可以是类

```
new Promise((resolve, reject) => {
  resolve(
    class {
      static then(resolve, reject) {
        setTimeout(() => {
          resolve("解决状态");
        }, 2000);
      }
    }
  );
}).then(
  v => {
    console.log(`fulfilled: ${v}`);
  },
  v => {
    console.log(`rejected: ${v}`);
  }
);

```

如果对象中的 then 不是函数，则将对象做为值传递

```
new Promise((resolve, reject) => {
  resolve();
})
.then(() => {
  return {
    then: "后盾人"
  };
})
.then(v => {
  console.log(v); //{then: "后盾人"}
});

```

#catch

下面使用未定义的变量同样会触发失败状态

```
let promise = new Promise((resolve, reject) => {
  hd;
}).then(
  value => console.log(value),
  reason => console.log(reason)
);

```

如果 onFulfilled 或 onRejected 抛出异常，则 p2 拒绝执行并返回拒因

```
let promise = new Promise((resolve, reject) => {
  throw new Error("fail");
});
let p2 = promise.then();
p2.then().then(null, resolve => {
  console.log(resolve + ",后盾人");
});

```

catch 用于失败状态的处理函数，等同于 `then(null,reject){}`

- 建议使用

catch 处理错误

- 将

catch 放在最后面用于统一处理前面发生的错误

```
const promise = new Promise((resolve, reject) => {
  reject(new Error("Notice: Promise Exception"));
}).catch(msg => {
  console.error(msg);
});

```

`catch` 可以捕获之前所有 `promise` 的错误，所以建议将 `catch` 放在最后。下例中 `catch` 也可以捕获到了第一个 `then` 返回 的 `promise` 的错误。

```
new Promise((resolve, reject) => {
  resolve();
```

```
})
.then(() => {
  return new Promise((resolve, reject) => {
    reject(".then ");
  });
})
.then(() => {})
.catch(msg => {
  console.log(msg);
});

```

错误是冒泡的操作的，下面没有任何一个`then`定义第二个函数，将一直冒泡到`catch`处理错误

```
new Promise((resolve, reject) => {
  reject(new Error("请求失败"));
})
.then(msg => {})
.then(msg => {})
.catch(error => {
  console.log(error);
});

```

`catch`也可以捕获对`then`抛出的错误处理

```
new Promise((resolve, reject) => {
  resolve();
})
.then(msg => {
  throw new Error("这是then 抛出的错误");
})
.catch(() => {
  console.log("33");
});

```

`catch`也可以捕获其他错误，下面在`then`中使用了未定义的变量，将会把错误抛出到`catch`

```
new Promise((resolve, reject) => {
  resolve("success");
})
.then(msg => {
  console.log(a);
})
.catch(reason => {
  console.log(reason);
});

```

#使用建议

建议将错误要交给`catch`处理而不是在`then`中完成，不建议使用下面的方式管理错误

```
new Promise((resolve, reject) => {
  reject(new Error("请求失败"));
}).then(
  msg => {
    console.log(msg);
  },
  error => {
    console.log(error);
  }
);

```

#处理机制

在 `promise` 中抛出的错误也会被`catch` 捕获

```
const promise = new Promise((resolve, reject) => {
  throw new Error("fail");
}).catch(msg => {
  console.log(msg.()+"后盾人");
});

```

可以将上面的理解为如下代码，可以理解为内部自动执行 `try...catch`

```
const promise = new Promise((resolve, reject) => {
  try {
    throw new Error("fail");
  } catch (error) {
    reject(error);
  }
}).catch(msg => {
  console.log(msg.());
});

```

但像下面的在异步中 `throw` 将不会触发 `catch`，而使用系统错误处理

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    throw new Error("fail");
  }, 2000);
}).catch(msg => {
  console.log(msg + "后盾人");
});

```

下面在`then` 方法中使用了没有定义的`hd`函数，也会抛除到 `catch` 执行，可以理解为内部自动执行 `try...catch`

```
const promise = new Promise((resolve, reject) => {
  resolve();
})
.then(() => {
  hd();
})
.catch(msg => {
  console.log(msg.());
});

```

在 `catch` 中发生的错误也会抛给最近的错误处理

```
const promise = new Promise((resolve, reject) => {
  reject();
})
.catch(msg => {
  hd();
})
.then(null, error => {
  console.log(error);
});

```

#定制错误

可以根据不同的错误类型进行定制操作，下面将参数错误与 404 错误分别进行了处理

```
class ParamError extends Error {
  constructor(msg) {
    super(msg);
    this.name = "ParamError";
  }
}
class HttpError extends Error {
  constructor(msg) {
    super(msg);
    this.name = "HttpError";
  }
}
function ajax(url) {
  return new Promise((resolve, reject) => {
    if (!/^http/.test(url)) {
      throw new ParamError("请求地址格式错误");
    }
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.onload = function() {
      if (this.status == 200) {
```

```
        resolve(JSON.parse(this.response));
      } else if (this.status == 404) {
        // throw new HttpError("用户不存在");
        reject(new HttpError("用户不存在"));
      } else {
        reject("加载失败");
      }
    };
    xhr.onerror = function() {
      reject(this);
    };
  });
}

ajax(`
http://localhost:8888/php/user.php?name=
ajax(`
.then(value => {
  console.log(value);
})
.catch(error => {
  if (error instanceof ParamError) {
    console.log(error.message);
  }
  if (error instanceof HttpError) {
    alert(error.message);
  }
  console.log(error);
});

```

#事件处理

unhandledrejection事件用于捕获到未处理的 Promise 错误，下面的 then 产生了错误，但没有 `catch` 处理，这时就会触发事件。该事件有可能在以后被废除，处理方式是对没有处理的错误直接终止。

```
window.addEventListener("unhandledrejection", function(event) {
  console.log(event.promise); // 产生错误的promise对象
  console.log(event.reason); // Promise的reason
```

```
});

new Promise((resolve, reject) => {
  resolve("success");
}).then(msg => {
  throw new Error("fail");
});

```

#finally

无论状态是`resolve` 或 `reject` 都会执行此动作，`finally` 与状态无关。

```
const promise = new Promise((resolve, reject) => {
  reject("hdcms");
})
.then(msg => {
  console.log("resolve");
})
.catch(msg => {
  console.log("reject");
})
.finally(() => {
  console.log("resolve/reject状态都会执行");
});

```

下面使用 `finally` 处理加载状态，当请求完成时移除加载图标。请在后台 php 文件中添加 `sleep(2);` 设置延迟响应

```
<body>
  <style>
    div {
      width: 100px;
      height: 100px;
      background: red;
      color: white;
      display: none;
    }
```

```
  </style>
 <div>loading...</div>
</body>
<script>
function ajax(url) {
  return new Promise((resolve, reject) => {
    document.querySelector("div").style.display = "block";
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(JSON.parse(this.response));
      } else {
        reject(this);
      }
    };
  });
}

ajax("
http://localhost:8888/php/user.php?name=
ajax("
  .then(user => {
    console.log(user);
  })
  .catch(error => {
    console.log(error);
  })
  .finally(() => {
    document.querySelector("div").style.display = "none";
  })
</script>

```

#实例操作

#异步请求

下面是将 `ajax` 修改为 `promise` 后，代码结构清晰了很多

```
function ajax(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(JSON.parse(this.response));
      } else {
        reject(this);
      }
    };
  });
}

ajax("
http://localhost:8888/php/user.php?name=
ajax("
.then(user =>ajax(`
http://localhost:8888/php/houdunren.php?id=
.then(user =>ajax(`
.then(lesson => {
  console.log(lesson);
});

```

#图片加载

下面是异步加载图片示例

```
function loadImage(file) {
  return new Promise((resolve, reject) => {
    const image = new Image();
    image.src = file;
    image.onload = () => {
      resolve(image);
    };
    image.onerror = reject;
    document.body.appendChild(image);
```

```
  });
}

loadImage("images/houdunren.png").then(image => {
  image.style.border = "solid 20px black";
  console.log("宽度:" + window.getComputedStyle(image).width);
});

```

#定时器

下面是封装的`timeout`函数，使用定时器操作更加方便

```
function timeout(times) {
  return new Promise(resolve => {
    setTimeout(resolve, times);
  });
}

timeout(3000)
  .then(() => {
    console.log("3秒后执行");
    return timeout(1000);
  })
  .then(() => {
    console.log("执行上一步的promise后1秒执行");
  });

```

封闭`setInterval`定时器并实现动画效果

```
<body>
  <style>
    div {
      width: 100px;
      height: 100px;
      background: yellowgreen;
      position: absolute;
    }
```

```
  </style>
  <div></div>
</body>
<script>
  function interval(delay = 1000, callback) {
    return new Promise(resolve => {
      let id = setInterval(() => {
        callback(id, resolve);
      }, delay);
    });
  }
  interval(100, (id, resolve) => {
    const div = document.querySelector("div");
    let left = parseInt(window.getComputedStyle(div).left);
    div.style.left = left + 10 + "px";
    if (left >= 200) {
      clearInterval(id);
      resolve(div);
    }
  }).then(div => {
    interval(50, (id, resolve) => {
      let width = parseInt(window.getComputedStyle(div).width);
      div.style.width = width - 10 + "px";
      if (width <= 20) {
        clearInterval(id);
      }
    });
  });
</script>

```

#链式操作

- 第个

then 都是一个 promise

- 如果

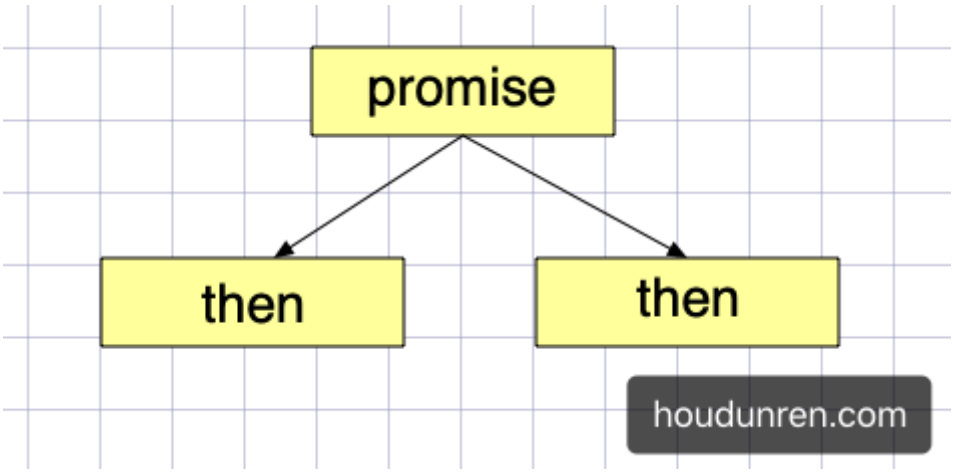
then 返回 promse，只当

promise 结束后，才会继承执行下一个

then

#语法介绍

下面是对同一个 `promise` 的多个 `then` ， 每个`then` 都得到了同一个 promise 结果，这不是链式操作，实际使用意义不大。




```
const promise = new Promise((resolve, reject) => {
  resolve("后盾人");
});
promise.then(hd => {
  hd += "-hdcms";
  console.log(hd); //后盾人-hdcms
});
promise.then(hd => {
  hd += "-houdunren";
  console.log(hd); //后盾人-houdunren
});

```

第一个 `then` 也是一个 promise，当没接受到结果是状态为 `pending`

```
const promise = new Promise((resolve, reject) => {
  resolve("后盾人");
});

console.log(
  promise.then(hd => {
    hd += "-hdcms";
    console.log(hd);
```

```
  })
); //Promise {<pending>}

```

`promise` 中的 `then` 方法可以链接执行，`then` 方法的返回值会传递到下一个`then` 方法。

- then 会返回一个

promise ，所以如果有多个

then 时会连续执行

- then 返回的值会做为当前

promise 的结果

下面是链式操作的 `then`，即始没有 `return` 也是会执行，因为每个`then` 会返回`promise`

```
new Promise((resolve, reject) => {
  resolve("后盾人");
})
.then(hd => {
  hd += "-hdcms";
  console.log(hd); //后盾人-hdcms
  return hd;
})
.then(hd => {
  hd += "-houdunren";
  console.log(hd); //后盾人-hdcms-houdunren
});

```

`then` 方法可以返回一个`promise` 对象，等`promise` 执行结束后，才会继承执行后面的 `then`。后面的 `then` 方法就是对新返回的`promise` 状态的处理

```
new Promise((resolve, reject) => {
  resolve("第一个promise");
})
.then(msg => {
  console.log(msg);
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("第二个promise");
    }, 3000);
  });
```

```
})
.then(msg => {
  console.log(msg);
});

```

#链式加载

使用`promise` 链式操作重构前面章节中的文件加载，使用代码会变得更清晰

```
function load(file) {
  return new Promise((resolve, reject) => {
    const script = document.createElement("script");
    script.src = file;
    script.onload = () => resolve(script);
    script.onerror = () => reject();
    document.body.appendChild(script);
  });
}

load("js/hd.js")
.then(() => load("js/houdunren.js"))
.then(() => houdunren());

```

#操作元素

下面使用 `promise` 对元素事件进行处理

```
<body>
  <div>
    <h2>第九章 闭包与作用域</h2>
    <button>收藏课程</button>
  </div>
</body>

<script>
new Promise(resolve => {
  document.querySelector("button").addEventListener("click", e => {
```

```
    resolve();
  });
})
.then(() => {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("执行收藏任务");
      resolve();
    }, 2000);
  });
})
.then(() => {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("更新积分");
      resolve();
    }, 2000);
  });
})
.then(() => {
  console.log("收藏成功! 奖励10积分");
})
.catch(error => console.log(errro));

```

#异步请求

下面是使用链式操作获取学生成绩

```
function ajax(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(JSON.parse(this.response));
      } else {
        reject(this);
```

```
      }
    };
  });
}
ajax("
http://localhost:8888/php/user.php?name=
ajax("
.then(user => {
  return ajax(`
http://localhost:8888/php/houdunren.php?id=
  return ajax(`
})
.then(lesson => {
  console.log(lesson);
});

```

#扩展接口

#resolve

使用 `promise.resolve` 方法可以快速的返回一个 promise 对象

根据值返加 `promise`

```
Promise.resolve("后盾人").then(value => {
  console.log(value); //后盾人
});

```

下面将请求结果缓存，如果再次请求时直接返回带值的 `promise`

- 为了演示使用了定时器，你也可以在后台设置延迟响应

```
function query(name) {
  const cache = query.cache || (query.cache = new Map());
  if (cache.has(name)) {
    console.log("走缓存了");
    return Promise.resolve(cache.get(name));
  }
  return ajax(`
http://localhost:8888/php/user.php?name=
```

```
    return ajax(`
      response => {
        cache.set(name, response);
        console.log("没走缓存");
        return response;
      }
    );
}
query("向军").then(response => {
  console.log(response);
});
setTimeout(() => {
  query("向军").then(response => {
    console.log(response);
  });
}, 1000);

```

如果是 `thenable` 对象，会将对象包装成 promise 处理，这与其他 promise 处理方式一样的

```
const hd = {
  then(resolve, reject) {
    resolve("后盾人");
  }
};
Promise.resolve(hd).then(value => {
  console.log(value);
});

```

#reject

和 `Promise.resolve` 类似，`reject` 生成一个失败的`promise`

```
Promise.reject("fail").catch(error => console.log(error));

```

下面使用 `Project.reject` 设置状态

```
new Promise(resolve => {
  resolve("后盾人");
})
.then(v => {
  if (v != "houdunren.com") return Promise.reject(new Error("fail"));
})
.catch(error => {
  console.log(error);
});

```

#all

使用`Promise.all`方法可以同时执行多个并行异步操作，比如页面加载时同进获取课程列表与推荐课程。

- 任何一个

Promise 执行失败就会调用

catch方法

- 适用于一次发送多个异步操作
- 参数必须是可迭代类型，如 Array/Set
- 成功后返回

promise 结果的有序数组

下例中当`hdcms、houdunren`两个 Promise 状态都为`fulfilled`时，hd 状态才为`fulfilled`。

```
const hdcms = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("第一个Promise");
  }, 1000);
});
const houdunren = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("第二个异步");
  }, 1000);
});
const hd = Promise.all([hdcms, houdunren])
  .then(results => {
    console.log(results);
```

```
  })
  .catch(msg => {
    console.log(msg);
  });

```

根据用户名获取用户，有任何一个用户获取不到时 `promise.all` 状态失败，执行 `catch` 方法

```
function ajax(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(JSON.parse(this.response));
      } else {
        reject(this);
      }
    };
  });
}

const api = "
http://localhost:8888/php
const api = "
const promises = ["向军", "后盾人"].map(name => {
  return ajax(`${api}/user.php?name=${name}`);
});

Promise.all(promises)
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    console.log(error);
  });

```

可以将其他非`promise` 数据添加到 `all` 中，它将被处理成 `Promise.resolve`

```
...
const promises = [
  ajax(`${api}/user.php?name=向军`),
  ajax(`${api}/user.php?name=后盾人`),
  { id: 3, name: "hdcms", email: "admin@hdcms.com" }
];
...

```

如果某一个`promise`没有 catch 处理，将使用`promise.all` 的 catch 处理

```
let p1 = new Promise((resolve, reject) => {
  resolve("fulfilled");
});
let p2 = new Promise((resolve, reject) => {
  reject("rejected");
});
Promise.all([p1, p2]).catch(reason => {
  console.log(reason);
});

```

#allSettled

`allSettled` 用于处理多个`promise` ，只关注执行完成，不关注是否全部执行成功，`allSettled` 状态只会是`fulfilled`。

下面的 p2 返回状态为 `rejected` ，但`promise.allSettled` 不关心，它始终将状态设置为 `fulfilled` 。

```
const p1 = new Promise((resolve, reject) => {
  resolve("resolved");
});
const p2 = new Promise((resolve, reject) => {
  reject("rejected");
});
Promise.allSettled([p1, p2])
.then(msg => {
  console.log(msg);
})
```

```
11
```

下面是获取用户信息，但不关注某个用户是否获取不成功

```
const api = "
http://localhost:8888/php
const api = "
const promises = [
  ajax(`${api}/user.php?name=向军`),
  ajax(`${api}/user.php?name=后盾人`)
];
Promise.allSettled(promises).then(response => {
  console.log(response);
});

```

#race

使用`Promise.race()`处理容错异步，和`race`单词一样哪个 Promise 快用哪个，哪个先返回用哪个。

- 以最快返回的 promise 为准
- 如果最快返加的状态为

rejected 那整个

promise为

rejected执行 cache

- 如果参数不是 promise，内部将自动转为 promise

下面将第一次请求的异步时间调整为两秒，这时第二个先返回就用第二人。

```
const hdcms = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("第一个Promise");
  }, 2000);
});
const houdunren = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("第二个异步");
  }, 1000);
});
Promise.race([hdcms, houdunren])
.then(results => {
```

```
  console.log(results);
})
.catch(msg => {
  console.log(msg);
});

```

获取用户资料，如果两秒内没有结果 `promise.race` 状态失败，执行`catch` 方法

```
const api = "
http://localhost:8888/php
const api = "
const promises = [
  ajax(`${api}/user.php?name=向军`),
  new Promise((a, b) =>
    setTimeout(() => b(new Error("request fail")), 2000)
  )
];
Promise.race(promises)
.then(response => {
  console.log(response);
})
.catch(error => {
  console.log(error);
});

```

#任务队列

#实现原理

如果 `then` 返回`promise` 时，后面的`then` 就是对返回的 `promise` 的处理

```
let promise = Promise.resolve();
let p1 = promise.then(() => {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`p1`);
      resolve();
```

```
    }, 1000);
  });
});
p1.then(() => {
  return new Promise((a, b) => {
    setTimeout(() => {
      console.log(`p2`);
    }, 1000);
  });
});

```

下面使用 `map` 构建的队列，有以下几点需要说明

- then 内部返回的

promise 更改外部的

promise 变量

- 为了让任务继承，执行完任务需要将

promise 状态修改为

fulfilled

```
function queue(nums) {
  let promise = Promise.resolve();
  nums.map(n => {
    promise = promise.then(v => {
      return new Promise(resolve => {
        console.log(n);
        resolve();
      });
    });
  });
}

queue([1, 2, 3, 4, 5]);

```

下面再来通过 `reduce` 来实现队列

```
function queue(nums) {
  return nums.reduce((promise, n) => {
```

```
    return promise.then(() => {
      return new Promise(resolve => {
        console.log(n);
        resolve();
      });
    });
  }, Promise.resolve());
}

queue([1, 2, 3, 4, 5]);

```

#队列请求

下面是异步加载用户并渲染到视图中的队列实例

- 请在后台添加延迟脚本，以观察队列执行过程
- 也可以在任何

promise 中添加定时器观察

```
class User {
 //加载用户
  ajax(user) {
    let url = `
http://localhost:8888/php/user.php?name=
    let url = `
    return new Promise(resolve => {
      let xhr = new XMLHttpRequest();
      xhr.open("GET", url);
      xhr.send();
      xhr.onload = function() {
        if (this.status == 200) {
          resolve(JSON.parse(this.response));
        } else {
          reject(this);
        }
      };
    });
  }
  //启动
  render(users) {
```

```
    users.reduce((promise, user) => {
      return promise
        .then(() => {
          return this.ajax(user);
        })
        .then(user => {
          return this.view(user);
        });
    }, Promise.resolve());
  }
  //宣染视图
  view(user) {
    return new Promise(resolve => {
      let h1 = document.createElement("h1");
      h1.innerHTML = user.name;
      document.body.appendChild(h1);
      resolve();
    });
  }
}
new User().render(["向军", "后盾人"]);

```

#高可用封装

上例中处理是在队列中完成，不方便业务定制，下面将 Promise 处理在剥离到外部

后台请求处理类

```
export default function(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest()
    xhr.open('GET', url)
    xhr.send()
    xhr.onload = function() {
      if (this.status === 200) {
        resolve(this.response)
      } else {
        reject(this)
      }
```

```
    }
  })
}

```

队列处理类

```
export default function(promises) {
  promises.reduce((promise, next) => promise.then(next), Promise.resolve())
}

```

后台脚本

```
<?php
$users = [
    1 => "小明",
    2 => "李四",
    3 => "张三"
];
sleep(1);
echo $users[$_GET['id']];

```

使用队列

```
<script type="module">
  import queue from './queue.js'
  import axios from './axios.js'
  queue(
    [1, 2, 3].map(v => () =>
      axios(`user.php?id=${v}`).then(user => console.log(user))
    )
  )
</script>

```

#async/await

使用 `async/await` 是 promise 的语法糖，可以让编写 promise 更清晰易懂，也是推荐编写 promise 的方式。

- async/await 本质还是 promise，只是更简洁的语法糖书写
- async/await 使用更清晰的 promise 来替换 promise.then/catch 的方式

#async

下面在 `hd` 函数前加上 async，函数将返回 promise，我们就可以像使用标准 Promise 一样使用了。

```
async function hd() {
  return "houdunren.com";
}
console.log(hd());
hd().then(value => {
  console.log(value);
});

```

如果有多个 await 需要排队执行完成，我们可以很方便的处理多个异步队列

```
async function hd(message) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(message);
    }, 2000);
  });
}
async function run() {
  let h1 = await hd("后盾人");
  console.log(h1);
  let h2 = await hd("houdunren.com");
  console.log(h2);
}
run();

```

#await

使用 `await` 关键词后会等待 promise 完

- await 后面一般是 promise，如果不是直接返回
- await 必须放在 async 定义的函数中使用
- await 用于替代

then 使编码更优雅

下例会在 await 这行暂停执行，直到等待 promise 返回结果后才继执行。

```
async function hd() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("houdunren.com");
    }, 2000);
  });
  let result = await promise;
  console.log(result);
}
hd()

```

一般 await 后面是外部其它的 promise 对象

```
async function hd() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("fulfilled");
    }, 2000);
  });
}
async function run() {
  let value = await hd();
  console.log("houdunren.com");
  console.log(value);
}
run();

```

下面是请求后台获取用户课程成绩的示例

```
async function user() {
```

```
  let user = await ajax(`
http://localhost:8888/php/user.php?name=
  let user = await ajax(`
  let lessons = await ajax(
    `
http://localhost:8888/php/houdunren.php?id=
    `
  );
  console.log(lessons);
}

```

也可以将操作放在立即执行函数中完成

```
(async () => {
  let user = await ajax(`
http://localhost:8888/php/user.php?name=
  let user = await ajax(`
  let lessons = await ajax(
    `
http://localhost:8888/php/houdunren.php?id=
    `
  );
  console.log(lessons);
})();

```

下面是使用 async 设置定时器，并间隔时间来输出内容

```
async function sleep(ms = 2000) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}
async function run() {
  for (const value of ["后盾人", "向军"]) {
    await sleep();
    console.log(value);
  }
}
run();
```

```
13
```

#加载进度

下面是请求后台加载用户并通过进度条展示的效果

```
<body>
  <style>
    div {
      height: 50px;
      width: 0px;
      background: green;
    }
  </style>
  <div id="loading"></div>
</body>
<script src="js/ajax.js"></script>
<script>
  async function query(name) {
    return ajax(`
http://localhost:8888/php/user.php?name=
    return ajax(`
  }
  (async () => {
    let users = ["后盾人", "向军", "李四", "王五", "赵六"];
    for (let i = 0; i < users.length; i++) {
      await query(users[i]);
      let progress = (i + 1) / users.length;
      loading.style.width = progress * 100 + "%";
    }
  })();
</script>

```

#类中使用

和 promise 一样，await 也可以操作`thenables`对象

```
class User {
```

```
  (name) {
    this.name = name;
  }
  then(resolve, reject) {
    let user = ajax(`
http://localhost:8888/php/user.php?name=
    let user = ajax(`
    resolve(user);
  }
}
async function get() {
  let user = await new User("向军");
  console.log(user);
}
get();

```

类方法也可以通过 `async` 与 `await` 来操作 promise

```
class User {
  () {}
  async get(name) {
    let user = await ajax(
      `
http://localhost:8888/php/user.php?name=
      `
    );
    user.name += "-houdunren.com";
    return user;
  }
}
new User().get("向军").then(resolve => {
  console.log(resolve);
});

```

#其他声明

函数声明

```
async function get(name) {
  return await ajax(`
http://localhost:8888/php/user.php?name=
  return await ajax(`
}
get("后盾人").then(user => {
  console.log(user);
});

```

函数表达式

```
let get = async function(name) {
  return await ajax(`
http://localhost:8888/php/user.php?name=
  return await ajax(`
};
get("后盾人").then(user => {
  console.log(user);
});

```

对象方法声明

```
let hd = {
  async get(name) {
    return await ajax(`
http://localhost:8888/php/user.php?name=
    return await ajax(`
  }
};

hd.get("后盾人").then(user => {
  console.log(user);
});

```

立即执行函数

```

```

```
(async () => {
  let user = await ajax(`
http://localhost:8888/php/user.php?name=
  let user = await ajax(`
  let lessons = await ajax(
    `
http://localhost:8888/php/houdunren.php?id=
    `
  );
  console.log(lessons);
})();

```

类方法中的使用

```
class User {
  async get(name) {
    return await ajax(`
http://localhost:8888/php/user.php?name=
    return await ajax(`
  }
}
let user = new User().get("后盾人").then(user => {
  console.log(user);
});

```

#错误处理

async 内部发生的错误，会将必变 promise 对象为 rejected 状态，所以可以使用`catch`来处理

```
async function hd() {
  console.log(houdunren);
}
hd().catch(error => {
  throw new Error(error);
});

```

下面是异步请求数据不存在时的错误处理

```
async function get(name) {
  return await ajax(`
http://localhost:8888/php/user.php?name=
  return await ajax(`
}

get("向军小哥").catch(error => {
  alert("用户不存在");
});

```

如果`promise` 被拒绝将抛出异常，可以使用 `try...catch` 处理错误

```
async function get(name) {
  try {
    let user = await ajax(
      `
http://localhost:8888/php/user.php?name=
      `
    );
    console.log(user);
  } catch (error) {
    alert("用户不存在");
  }
}
get("向军老师");

```

多个 await 时当前面的出现失败，后面的将不可以执行

```
async function hd() {
  await Promise.reject("fail");
  await Promise.resolve("success").then(value => {
    console.log(value);
  });
}
hd();

```

如果对前一个错误进行了处理，后面的 await 可以继续执行

```
async function hd() {
  await Promise.reject("fail").catch(e => console.log(e));
  await Promise.resolve("success").then(value => {
    console.log(value);
  });
}
hd();

```

也可以使用 `try...catch` 特性忽略不必要的错误

```
async function hd() {
  try {
    await Promise.reject("fail");
  } catch (error) {}
  await Promise.resolve("success").then(value => {
    console.log(value);
  });
}
hd();

```

也可以将多个 await 放在 try...catch 中统一处理错误

```
async function hd(name) {
  const host = "
http://localhost:8888/php
  const host = "
  try {
    const user = await ajax(`${host}/user.php?name=${name}`);
    const lessons = await ajax(`${host}/user.php?id=${user.id}`);
    console.log(lessons);
  } catch (error) {
    console.log("用户不存在");
  }
}
hd("后盾人教程");

```

#并发执行

有时需要多个 await 同时执行，有以下几种方法处理，下面多个 await 将产生等待

```
async function p1() {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("houdunren");
      resolve();
    }, 2000);
  });
}
async function p2() {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("hdcms");
      resolve();
    }, 2000);
  });
}
async function hd() {
  await p1();
  await p2();
}
hd();

```

使用 `Promise.all()` 处理多个 promise 并行执行

```
async function hd() {
  await Promise.all([p1(), p2()]);
}
hd();

```

让 promise 先执行后再使用 await 处理结果

```
async function hd() {
  let h1 = p1();
```

```
  let h2 = p2();
  await h1;
  await h2;
}
hd();
```