期约与异步函数

异步函数

ES8新增 async/await
旨在解决利用异步结构组织代码的问题

- async
  async关键字可以让函数具有异步特征
  异步函数return返回值会被Promise.resolve包装成Promise对象
  异步函数抛异常 需要.catch捕获
- await
  await关键字 暂停异步函数后面的代码。等待Promise解决
  限制，必须在异步函数中使用

停止和恢复执行

await后面跟Promise会更复杂（可以认为会等待更久一些，相对await后跟值）

```
async function foo() {
 console.log(await Promise.resolve('foo'));
}
async function bar() {
 console.log(await 'bar');
}
async function baz() {
 console.log('baz');
}
foo();
bar();
baz();
// baz
// bar
// foo
```

```
async function foo() {
 console.log(2);
 await null;
 console.log(4);
}
console.log(1);
foo();
console.log(3);
// 1
// 2
// 3
// 4
```

```
async function foo() {
 console.log(2);
 console.log(await Promise.resolve(8));
 console.log(9);
}
async function bar() {
 console.log(4);
 console.log(await 6);
 console.log(7);
}
console.log(1);
foo();
console.log(3);
bar();
console.log(5);
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

异步函数策略

在使用异步函数时，需要注意

实现sleep()

以前 setTimeout()
现在

```
async function sleep(delay) {
 return new Promise((resolve) => setTimeout(resolve, delay));
}
async function foo() {
 const t0 = Date.now();
 await sleep(1500); // 暂停约 1500 毫秒
 console.log(Date.now() - t0);
}
foo();
// 1502
```

利用平行执行

await不留心会错过平行加速

```
async function randomDelay(id) {
 // 延迟 0~1000 毫秒
 const delay = Math.random() * 1000;
 return new Promise((resolve) => setTimeout(() => {
 console.log(`${id} finished`);
```

```
 resolve();
 }, delay));
}
async function foo() {
 const t0 = Date.now();
 await randomDelay(0);
 await randomDelay(1);
 await randomDelay(2);
 await randomDelay(3);
 await randomDelay(4);
 console.log(`${Date.now() - t0}ms elapsed`);
}
foo();
// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 877ms elapsed
```

for 循环包装

```
 for (let i = 0; i < 5; ++i) {
 await randomDelay(i);
 }
```

如果没有依赖关系，就没必要全等待

```
 const p0 = randomDelay(0);
 const p1 = randomDelay(1);
 const p2 = randomDelay(2);
 const p3 = randomDelay(3);
 const p4 = randomDelay(4);
 await p0;
 await p1;
 await p2;
 await p3;
 await p4;
```

循环包装

```
const promises = Array(5).fill(null).map((_, i) => randomDelay(i));
 for (const p of promises) {
 await p;
 }
```

虽然期约没有按照顺序执行，但 await按顺序收到了每个期约的值

串行执行期约

加个 async 就可以直接用.then 串行了

栈追踪与内存管理

Promise与异步函数 内存表示差别很大

```
function fooPromiseExecutor(resolve, reject) {
    setTimeout(reject, 1000, 'bar');
}
function foo() {
    new Promise(fooPromiseExecutor);
}

foo();
// Uncaught (in promise) bar
// setTimeout
// setTimeout (async)
// fooPromiseExecutor
// foo


function fooPromiseExecutor(resolve, reject) {
 setTimeout(reject, 1000, 'bar');
}
async function foo() {
 await new Promise(fooPromiseExecutor);
}
foo();
// Uncaught (in promise) bar
// foo
// async function (async)
// foo
```

异步函数 栈追踪信息就准确地反映了当前的调用栈
没有额外的消耗，性能比Promise更好