一、promise 请求超时处理

```
// 一、promise 请求超时处理
// 需求：在微服务中发送一个请求，如果三秒钟还没有收到结果，我们就认为失败。

// 这里面是使用的是Promise.race。

// Promise.race()方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。并行执行

// const p = Promise.race([p1, p2, p3]);

// 上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。
// 那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。
// race
const p = Promise.race([
  getUrl("/resource-that-may-take-a-while"),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error("request timeout")), 3001);
  }),
]);

p.then(console.log).catch(console.error);
```

二、promise 三次重试

```
// 二、promise 三次重试
// 需求：用promise实现一个可以指定重试次数的方法，如果重试次数等于指定的重试测试时，还没有成功，则认为失败。

var getData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let num = Math.random();
      console.log(num);
      if (num > 1) {
        resolve();
      } else {
        reject();
      }
    }, 2000);
  });
};

function retryData(getData, times, delay) {
  return new Promise((resolve, reject) => {
```

```
    function attempt() {
      times--;
      getData()
        .then(resolve)
        .catch((erro) => {
          console.log(`还有 ${times} 次尝试`);
          if (times < 1) {
            reject(erro);
          } else {
            setTimeout(() => {
              attempt();
            }, delay * 1000);
          }
        });
    }
    attempt();
  });
}

retryData(getData, 3, 0.5)
  .then((res) => {
    console.error("执行成功", res);
  })
  .catch((err) => {
    console.error("尝试失败");
  });

// 0.2537473977539608
// 还有 2 次尝试
// 0.11891286863089512
// 还有 1 次尝试
// 0.27074469430076986
// 还有 0 次尝试
// 尝试失败
```

三、promise 并发请求并控制请求数目

```
// 三、promise并发请求并控制请求数目
// 有 n 个图片资源的 url，已经存储在数组 urls 中
//  (即urls = ['http://example.com/1.jpg', ….,
'http://example.com/8.jpg'])，
// 而且已经有一个函数 function loadImg，输入一个 url 链接，返回一个 Promise，
// 该 Promise 在图片下载完成的时候 resolve，下载失败则 reject。

// 但是我们要求，任意时刻，同时下载的链接数量不可以超过 3 个。
// 这里面的思想 我们主要使用async和await 来形成阻塞函数，
// 当我们的请求书大于3个的时候，我们把promise中的resolve保存到数组里面，先不执行。
// 当有图片加载出来后，我们在继续执行

var urls = [1, 2, 3, 4, 5, 6, 7, 8, 9];
```

```
function loadImg(url) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`第 ${url} 张图片加载完成`);
      resolve("success");
    }, 2 * 1000);
  });
}

// 计数器
var count = 0;
// 全局锁
var lock = [];
var l = urls.length;

// 阻塞函数
function block() {
  let _resolve;
  return new Promise((resolve, reject) => {
    _resolve = resolve;
    // resolve不执行,将其推入lock数组;
    lock.push(_resolve);
  });
}
// 叫号机
function next() {
  lock.length && lock.shift()();
}

async function reuqest() {
  if (count >= 3) {
    //超过限制利用await和promise进行阻塞;
    await block();
  }
  if (urls.length > 0) {
    console.log(count);
    count++;
    await loadImg(urls.shift());
    count--;
    next();
  }
}

for (let i = 0; i < l; i++) {
  reuqest();
}
// 0
// 1
// 2
// 第 1 张图片加载完成
// 2
// 第 2 张图片加载完成
// 2
```

```
// 第 3 张图片加载完成
// 2
// 第 4 张图片加载完成
// 2
// 第 5 张图片加载完成
// 2
// 第 6 张图片加载完成
// 2
// 第 7 张图片加载完成
// 第 8 张图片加载完成
// 第 9 张图片加载完成
```

四、promise 缓存图片

```
// 如果有巨量的图片要展示，除了懒加载的方式
// 有没有什么其他方法限制一下
// 同时加载图片数量
function limitLoad(urls, handler, limit) {
  const sequence = [].concat(urls);
  let promises = [];
  promises = sequence.splice(0, limit).map((url, index) => {
    return handler(url).then(() => index);
  });
  let p = Promise.race(promises);
  for (let i = 0; i < sequence.length; i++) {
    p = p
      .then((res) => {
        promises[res] = handler(sequence[i]).then(() => res);
        return Promise.race(promises);
      })
      .then()
      .then();
  }
}
const urls = [
  { info: "aaa" },
  { info: "bbb" },
  { info: "ccc" },
  { info: "ddd" },
  { info: "eee" },
  { info: "fff" },
  { info: "ggg" },
  { info: "hhh" },
];
function loadImg(url) {
  return new Promise((resolve, reject) => {
    console.log("----" + url.info + "   start!");
    setTimeout(() => {
      console.log(url.info + "   OK!");
      resolve();
```

```
      }, url.time);
    });
  }

  limitLoad(urls, loadImg, 3);
```