1. 什么是 Loader

在撸一个 loader 前，我们需要先知道它到底是什么。本质上来说，loader 就是一个 node 模块，这很符合 webpack 中「万物皆模块」的思路。既然是 node 模块，那就一定会导出点什么。在 webpack 的定义中，loader 导出一个函数，loader 会在转换源模块（resource）的时候调用该函数。在这个函数内部，我们可以通过传入 this 上下文给 Loader API 来使用它,因此我们也可以概括一下 loader 的功能：把源模块转换成通用模块。

2. Loader 怎么用

配置 webpack config 文件

- 单个 loader 的配置

```
let webpackConfig = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            //这里写 loader 的路径
            loader: path.resolve(__dirname, 'loaders/a-loader.js'),
            options: {
              /* ... */
            },
          },
        ],
      },
    ],
  },
};
```

- 多个 loader 的配置

```
let webpackConfig = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            //这里写 loader 名即可
            loader: 'a-loader',
            options: {
              /* ... */
            },
          },
          {
            loader: 'b-loader',
            options: {
              /* ... */
            },
          },
        ],
      },
    ],
  },
  resolveLoader: {
    // 告诉 webpack 该去那个目录下找 loader 模块
    modules: ['node_modules', path.resolve(__dirname, 'loaders')],
  },
};
```

3. 开发 loader 的注意项

了解了基本模式后，我们先不急着开发。所谓磨刀不误砍柴工，我们先看看开发一个 loader 需要注意些什么，这样可以少走弯路，提高开发质量。下面是 webpack 提供的几点指南，它们按重要程度排序，注意其中有些点只适用特定情况。

1. 单一职责
   一个 loader 只做一件事，这样不仅可以让 loader 的维护变得简单，还能让 loader 以不同的串联方式组合出符合场景需求的搭配。
2. 链式组合
   这一点是第一点的延伸。好好利用 loader 的链式组合的特型，可以收获意想不到的效果。具体来说，写一个能一次干 5 件事情的 loader ，不如细分成 5 个独立loader, 每个loader只能干一件事情。 也许其中几个能用在其他你暂时还没想到的场景。
   ::: tip
   事实上串联组合中的 loader 并不一定要返回 JS 代码。只要下游的 loader 能有效处理上游 loader

的输出，那么上游的 loader 可以返回任意类型的模块。
:::

3. 模块化
   保证 loader 是模块化的。loader 生成模块需要遵循和普通模块一样的设计原则。
4. 无状态
   在多次模块的转化之间，我们不应该在 loader 中保留状态。每个 loader 运行时应该确保与其他编译好的模块保持独立，同样也应该与前几个 loader 对相同模块的编译结果保持独立。
5. 使用 `Loader` 实用工具
   请好好利用 loader-utils 包，它提供了很多有用的工具，最常用的一个就是获取传入 loader 的 options。除了 loader-utils 之外,还有 schema-utils 包，我们可以用 schema-utils 提供的工具，获取用于校验 options 的 JSON Schema 常量，从而校验 loader options。下面给出的例子简要地结合了上面提到的两个工具包：

```
import { getOptions } from 'loader-utils';
import { validateOptions } from 'schema-utils';

const schema = {
  type: object,
  properties: {
    test: {
      type: string,
    },
  },
};

export default function(source) {
  const options = getOptions(this);

  validateOptions(schema, options, 'Example Loader');

  // 在这里写转换 source 的逻辑 ...
  return `export default ${JSON.stringify(source)}`;
}
```

理解 url-loader 的工作原理

核心代码部分

1. `index.js` 文件

```
import path from 'path';

import { getOptions } from 'loader-utils';
import { validate } from 'schema-utils';
import mime from 'mime-types';

import normalizeFallback from './utils/normalizeFallback';
import schema from './options.json';

// 判断是否需要转化成base64
function shouldTransform(limit, size) {
  if (typeof limit === 'boolean') {
    return limit;
  }

  if (typeof limit === 'string') {
    return size <= parseInt(limit, 10);
  }

  if (typeof limit === 'number') {
    return size <= limit;
  }

  return true;
}

// 返回文件类型，形如'application/json; charset=utf-8'|'text/html; charset=utf-8'
function getMimetype(mimetype, resourcePath) {
  if (typeof mimetype === 'boolean') {
    if (mimetype) {
      // 返回文件类型
      const resolvedMimeType = mime.contentType(path.extname(resourcePath));

      if (!resolvedMimeType) {
        return '';
      }

      return resolvedMimeType.replace(/;\s+charset/i, ';charset');
    }

    return '';
  }

  if (typeof mimetype === 'string') {
    return mimetype;
  }

  const resolvedMimeType = mime.contentType(path.extname(resourcePath));

  if (!resolvedMimeType) {
    return '';
```

```
  }

  return resolvedMimeType.replace(/;\s+charset/i, ';charset');
}

// 返回转码类型base64或者不处理
function getEncoding(encoding) {
  if (typeof encoding === 'boolean') {
    return encoding ? 'base64' : '';
  }

  if (typeof encoding === 'string') {
    return encoding;
  }

  return 'base64';
}

// 生成结果字符串
function getEncodedData(generator, mimetype, encoding, content, resourcePath) {
  if (generator) {
    return generator(content, mimetype, encoding, resourcePath);
  }

  return `data:${mimetype}${encoding ? `;${encoding}` : ''},${content.toString(
    // eslint-disable-next-line no-undefined
    encoding || undefined
  )}`;
}

// main，入口函数
export default function loader(content) {
  // Loader Options
  const options = getOptions(this) || {};

  validate(schema, options, {
    name: 'URL Loader', // 指定shame报错时的名称
    baseDataPath: 'options', //指定shame报错时的基路径
  });

  // No limit or within the specified limit
  if (shouldTransform(options.limit, content.length)) {
    const { resourcePath } = this;
    const mimetype = getMimetype(options.mimetype, resourcePath);
    const encoding = getEncoding(options.encoding);

    if (typeof content === 'string') {
      // eslint-disable-next-line no-param-reassign
      content = Buffer.from(content);
    }
```

```
    const encodedData = getEncodedData(
      options.generator,
      mimetype,
      encoding,
      content,
      resourcePath
    );

    const esModule =
      typeof options.esModule !== 'undefined' ? options.esModule : true;

    return `${
      esModule ? 'export default' : 'module.exports ='
    } ${JSON.stringify(encodedData)}`;
  }

  // Normalize the fallback.
  // 定义超出大小的文件用啥loader处理
  const {
    loader: fallbackLoader,
    options: fallbackOptions,
  } = normalizeFallback(options.fallback, options);

  // Require the fallback.
  // eslint-disable-next-line global-require, import/no-dynamic-require
  const fallback = require(fallbackLoader);

  // Call the fallback, passing a copy of the loader context. The copy has the query replaced. T
  // loader receives the query which was intended for it instead of the query which was intendec
  const fallbackLoaderContext = Object.assign({}, this, {
    query: fallbackOptions,
  });

  return fallback.call(fallbackLoaderContext, content);
}

// Loader Mode
export const raw = true;
```

2. `normalizeFallback.js` 文件

```
import loaderUtils from 'loader-utils';

export default function normalizeFallback(fallback, originalOptions) {
  let loader = 'file-loader'; // 默认会使用file-loader处理
  let options = {};

  if (typeof fallback === 'string') {
    loader = fallback;

    const index = fallback.indexOf('?');

    if (index >= 0) {
      loader = fallback.substr(0, index);
      options = loaderUtils.parseQuery(fallback.substr(index));
    }
  }

  if (fallback !== null && typeof fallback === 'object') {
    ({ loader, options } = fallback);
  }

  options = Object.assign({}, originalOptions, options);

  delete options.fallback;

  return { loader, options };
}
```