コードブロックのサンプル

情報科学科
Rai

2024 年 5 月 14 日

Hello, world!

```rust
fn main() {
    println!("Hello, world!");
}
```

Custom Callout

```typst
#import "@local/jsreport:0.1.0": callout, create-callout

#create-callout(
  "spark",
  (
    "Spark",
    image.decode("<svg width=\"15\" height=\"15\" viewBox=\"0 0 15 15\" fill=\"none\" xmlns=\"http://www.w3.org/2000/svg\"><path d=\"M8.69667 0.0403541C8.90859 0.131038 9.03106 0.354857 8.99316 0.582235L8.0902 6.00001H12.5C12.6893 6.00001 12.8625 6.10701 12.9472 6.27641C13.0319 6.4458 13.0136 6.6485 12.8999 6.80001L6.89997 14.8C6.76167 14.9844 6.51521 15.0503 6.30328 14.9597C6.09135 14.869 5.96888 14.6452 6.00678 14.4178L6.90974 9H2.49999C2.31061 9 2.13748 8.893 2.05278 8.72361C1.96809 8.55422 1.98636 8.35151 2.09999 8.2L8.09997 0.200038C8.23828 0.0156255 8.48474 -0.0503301 8.69667 0.0403541ZM3.49999 8.00001H7.49997C7.64695 8.00001 7.78648 8.06467 7.88148 8.17682C7.97648 8.28896 8.01733 8.43723 7.99317 8.5822L7.33027 12.5596L11.5 7.00001H7.49997C7.353 7.00001 7.21347 6.93534 7.11846 6.8232C7.02346 6.71105 6.98261 6.56279 7.00678 6.41781L7.66968 2.44042L3.49999 8.00001Z\" fill=\"currentColor\" fill-rule=\"evenodd\" clip-rule=\"evenodd\"></path></svg>")
  )
)

#callout("spark")[
  Sparking!!
]
```

remark-callout

```ts
import { defu } from "defu";
import type { Properties } from "hast";
import type * as mdast from "mdast";
import type { Plugin } from "unified";
import { visit } from "unist-util-visit";
import type { VFile } from "vfile";

export type Options = OptionsBuilder<NodeOptions | NodeOptionsFunction>;

export type OptionsBuilder<N> = {
  /**
```

```
   * The root node of the callout.
   *
   * @default
   * (callout) => ({
   *   tagName: callout.isFoldable ? "details" : "div",
   *   properties: {
   *     dataCallout: true,
   *     dataCalloutType: callout.type,
   *     open: callout.defaultFolded === undefined ? false : !callout.defaultFolded,
   *   },
   * })
   */
  root?: N;

  /**
   * The title node of the callout.
   *
   * @default
   * (callout) => ({
   *   tagName: callout.isFoldable ? "summary" : "div",
   *   properties: {
   *     dataCalloutTitle: true,
   *   },
   * })
   */
  title?: N;

  /**
   * The body node of the callout.
   *
   * @default
   * () => ({
   *   tagName: "div",
   *   properties: {
   *     dataCalloutBody: true,
   *   },
   * })
   */
  body?: N;

  /**
   * A list of callout types that are supported.
   * - If `undefined`, all callout types are supported. This means that this plugin will not check if the given callout type is in `callouts` and never call `onUnknownCallout`.
   * - If a list, only the callout types in the list are supported. This means that if the given callout type is not in `callouts`, this plugin will call `onUnknownCallout`.
   * @example ["info", "warning", "danger"]
   * @default undefined
   */
  callouts?: string[] | null;

  /**
   * A function that is called when the given callout type is not in `callouts`.
   *
   * - If the function returns `undefined`, the callout is ignored. This means that the callout is rendered as a normal blockquote.
   * - If the function returns a `Callout`, the callout is replaced with the returned `Callout`.
   */
  onUnknownCallout?: (callout: Callout, file: VFile) => Callout | undefined;
};

export type NodeOptions = {
  /**
   * The HTML tag name of the node.
   *
   * @see https://github.com/syntax-tree/hast?tab=readme-ov-file#element
   */
  tagName: string;

  /**
   * The html properties of the node.
   *
   * @see https://github.com/syntax-tree/hast?tab=readme-ov-file#properties
   * @see https://github.com/syntax-tree/hast?tab=readme-ov-file#element
   * @example { "className": "callout callout-info" }
   */
  properties: Properties;
};

export type NodeOptionsFunction = (callout: Callout) => NodeOptions;

export const defaultOptions: Required<Options> = {
  root: (callout) => ({
    tagName: callout.isFoldable ? "details" : "div",
    properties: {
      dataCallout: true,
      dataCalloutType: callout.type,
      open:
        callout.defaultFolded === undefined ? false : !callout.defaultFolded,
    },
  }),
  title: (callout) => ({
    tagName: callout.isFoldable ? "summary" : "div",
    properties: {
      dataCalloutTitle: true,
    },
  }),
  body: () => ({
    tagName: "div",
    properties: {
      dataCalloutBody: true,
    },
  }),
  callouts: null,
  onUnknownCallout: () => undefined,
};

const initOptions = (options?: Options) => {
  const defaultedOptions = defu(options, defaultOptions);

  return Object.fromEntries(
    Object.entries(defaultedOptions).map(([key, value]) => {
```

```typescript
      if (
        ["root", "title", "body"].includes(key) &&
        typeof value !== "function"
      )
        return [key, () => value];

      return [key, value];
    }),
  ) as Required<OptionsBuilder<NodeOptionsFunction>>;
};

/**
 * A remark plugin to parse callout syntax.
 */
export const remarkCallout: Plugin<[Options?], mdast.Root> = (_options) => {
  const options = initOptions(_options);

  return (tree, file) => {
    visit(tree, "blockquote", (node) => {
      const paragraphNode = node.children[0];
      if (paragraphNode.type !== "paragraph") return;

      const calloutTypeTextNode = paragraphNode.children[0];
      if (calloutTypeTextNode.type !== "text") return;

      // Parse callout syntax
      // e.g. "[!note] title"
      const [calloutTypeText, ...calloutBodyText] =
        calloutTypeTextNode.value.split("\n");
      const calloutData = parseCallout(calloutTypeText);
      if (calloutData == null) return;
      if (
        options.callouts != null &&
        !options.callouts.includes(calloutData.type)
      ) {
        const newCallout = options.onUnknownCallout(calloutData, file);
        if (newCallout == null) return;

        calloutData.type = newCallout.type;
        calloutData.isFoldable = newCallout.isFoldable;
        calloutData.title = newCallout.title;
      }

      // Generate callout root node
      node.data = {
        ...node.data,
        hName: options.root(calloutData).tagName,
        hProperties: {
          // @ts-ignore error TS2339: Property 'hProperties' does not exist on type
'BlockquoteData'.
          ...node.data?.hProperties,
          ...options.root(calloutData).properties,
        },
      };

      // Generate callout body node
      const bodyNode: (mdast.BlockContent | mdast.DefinitionContent)[] = [
        {
          type: "paragraph",
          children: [],
        },
        ...node.children.splice(1),
      ];
      if (bodyNode[0].type !== "paragraph") return; // type check
      if (calloutBodyText.length > 0) {
        bodyNode[0].children.push({
          type: "text",
          value: calloutBodyText.join("\n"),
        });
      }

      // Generate callout title node
      const titleNode: mdast.Paragraph = {
        type: "paragraph",
        data: {
          hName: options.title(calloutData).tagName,
          hProperties: {
            ...options.title(calloutData).properties,
          },
        },
        children: [],
      };
      if (calloutData.title != null) {
        titleNode.children.push({
          type: "text",
          value: calloutData.title,
        });
      }
      if (calloutBodyText.length <= 0) {
        for (const [i, child] of paragraphNode.children.slice(1).entries()) {
          // All inline node before the line break is added as callout title
          if (child.type !== "text") {
            titleNode.children.push(child);
            continue;
          }

          // Add the part before the line break as callout title and the part after as callout body
          const [titleText, ...bodyTextLines] = child.value.split("\n");
          if (titleText) {
            // Add the part before the line break as callout title
            titleNode.children.push({
              type: "text",
              value: titleText,
            });
          }
          if (bodyTextLines.length > 0) {
            // Add the part after the line break as callout body
            if (bodyNode[0].type !== "paragraph") return;
            bodyNode[0].children.push({
              type: "text",
              value: bodyTextLines.join("\n"),
            });
            // Add all nodes after the current node as callout body
```

```
              bodyNode[0].children.push(...paragraphNode.children.slice(i + 2));
              break;
            }
          }
        } else {
          // Add all nodes after the current node as callout body
          bodyNode[0].children.push(...paragraphNode.children.slice(1));
        }

        // Add body and title to callout root node children
        node.children = [
          titleNode,
          {
            type: "blockquote",
            data: {
              hName: options.body(calloutData).tagName,
              hProperties: {
                ...options.body(calloutData).properties,
              },
            },
            children: bodyNode,
          },
        ];
      });
    };
  };

  export type Callout = {
    /**
     * The type of the callout.
     */
    type: string;

    /**
     * Whether the callout is foldable.
     */
    isFoldable: boolean;

    /**
     * Whether the callout is folded by default.
     */
    defaultFolded?: boolean;

    /**
     * The title of the callout.
     */
    title?: string;
  };

  /**
   * @example
   * \`\`\`
   * const callout = parseCallout("[!info]");  // { type: "info", isFoldable: false, title:
undefined }
   * const callout = parseCallout("[!info");   // undefined
   * \`\`\`
   */
```

```typescript
export const parseCallout = (
  text: string | null | undefined,
): Callout | undefined => {
  if (text == null) return;

  const match = text.match(
    /^\[!(?<type>.+?)\](?<isFoldable>[-+])?\s?(?<title>.+)?$/,
  );
  if (match?.groups?.type == null) return undefined;

  return {
    type: match.groups.type,
    isFoldable: match.groups.isFoldable != null,
    defaultFolded:
      match.groups.isFoldable == null
        ? undefined
        : match.groups.isFoldable === "-",
    title: match.groups.title,
  };
};
```