../redux-saga/packages/babel-plugin-redux-saga/src/index.js

```
var SourceMapConsumer = require('source-map').SourceMapConsumer
var pathFS = require('path')

var symbolName = '@@redux-saga/LOCATION'

function getSourceCode(path) {
  // use `toString` for babel v7, `getSource` for older versions
  const rawCode = Object.prototype.hasOwnProperty.call(path, 'toString') ? path.toString() : path.getSource()
  return rawCode.replace(/^(yield\*?)\s+/, '')
}

function getFilename(fileOptions, useAbsolutePath) {
  if (useAbsolutePath) {
    return fileOptions.filename
  }
  // babel v7 defines cwd. for v6 use fallback
  const cwd = fileOptions.cwd || process.cwd()
  return pathFS.relative(cwd, fileOptions.filename)
}

function isSaga(path) {
  return path.node.generator
}

module.exports = function (babel) {
  var { types: t, template } = babel
  var sourceMap = null
  var alreadyVisited = new WeakSet()

  var extendExpressionWithLocationTemplate = template(`
    Object.defineProperty(TARGET, SYMBOL_NAME, {
      value: {
        fileName: FILENAME,
        lineNumber: LINE_NUMBER,
        code: SOURCE_CODE,
      },
    });
  `)

  /**
   *  Genetares location descriptor
   */

  function createLocationExtender(node, location, sourceCode) {
    const extendExpressionWithLocation = extendExpressionWithLocationTemplate({
      TARGET: node,
      SYMBOL_NAME: t.stringLiteral(symbolName),
      FILENAME: t.stringLiteral(location.fileName),
      LINE_NUMBER: t.numericLiteral(location.lineNumber),
      SOURCE_CODE: sourceCode ? t.stringLiteral(sourceCode) : t.nullLiteral(),
    })

    return extendExpressionWithLocation.expression
  }

  function calcLocation(loc, fileName) {
    var lineNumber = loc.start.line

    if (!sourceMap) {
      return {
        lineNumber: lineNumber,
        fileName: fileName,
      }
    }
    var mappedData = sourceMap.originalPositionFor({
      line: loc.start.line,
      column: loc.start.column,
    })

    return {
      lineNumber: mappedData.line,
      fileName: fileName + ' (' + mappedData.source + ')',
    }
  }

  var visitor = {
    Program: function (path, state) {
      // clean up state for every file
      sourceMap = state.file.opts.inputSourceMap ? new SourceMapConsumer(state.file.opts.inputSourceMap) : null
    },
    /**
     * attach location info object to saga
     *
     * @example
     * input
     *  function * effectHandler(){}
     * output
     *  function * effectHandler(){}
     *  Object.defineProperty(effectHandler, "@@redux-saga/LOCATION", {
     *    value: { fileName: ..., lineNumber: ... }
     *  })
     */
    FunctionDeclaration(path, state) {
      var node = path.node
      if (!node.loc || !isSaga(path)) return

      var functionName = node.id.name
      var filename = getFilename(state.file.opts, state.opts.useAbsolutePath)

      var locationData = calcLocation(node.loc, filename)

      const extendedDeclaration = createLocationExtender(t.identifier(functionName), locationData)

      // https://github.com/babel/babel/issues/4007
      if (path.parentPath.isExportDefaultDeclaration() || path.parentPath.isExportDeclaration()) {
        path.parentPath.insertAfter(extendedDeclaration)
      } else {
```

```
        path.insertAfter(extendedDeclaration)
      }
    },
    FunctionExpression(path, state) {
      var node = path.node
      if (!node.loc || !isSaga(path) || alreadyVisited.has(node)) return
      alreadyVisited.add(node)

      var filename = getFilename(state.file.opts, state.opts.useAbsolutePath)

      var locationData = calcLocation(node.loc, filename)
      var sourceCode = getSourceCode(path)

      const extendedExpression = createLocationExtender(node, locationData, sourceCode)

      path.replaceWith(extendedExpression)
    },
    /**
     * attach location info object to effect descriptor
     * ignores delegated yields
     *
     * @example
     * input
     *  yield call(smthelse)
     * output
     *  yield (function () {
     *    return Object.defineProperty(test1, "@@redux-saga/LOCATION", {
     *      value: { fileName: ..., lineNumber: ... }
     *    })
     *  })()
     */
    YieldExpression(path, state) {
      var node = path.node
      var yielded = node.argument
      if (!node.loc || node.delegate) return
      if (!t.isCallExpression(yielded) && !t.isLogicalExpression(yielded)) return

      var filename = getFilename(state.file.opts, state.opts.useAbsolutePath)

      var locationData = calcLocation(node.loc, filename)
      var sourceCode = getSourceCode(path)

      node.argument = createLocationExtender(yielded, locationData, sourceCode)
    },
  }

  return {
    visitor,
  }
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration/babel6-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/declaration/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration/source.js",
    lineNumber: 1,
    code: null
  }
})

function* test2() {
  yield 2;
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration/babel7-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/declaration/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration/source.js",
    lineNumber: 1,
    code: null
  }
})
```

```
function* test2() {
  yield 2;
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-es6-modules/babel6-expected.js

```
export function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/declaration-es6-modules/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}
Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-es6-modules/source.js",
    lineNumber: 1,
    code: null
  }
})
export default function* test2() {
  yield 2;
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-es6-modules/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-es6-modules/babel7-expected.js

```
export function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/declaration-es6-modules/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-es6-modules/source.js",
    lineNumber: 1,
    code: null
  }
})
export default function* test2() {
  yield 2;
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-es6-modules/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-es6-modules/source.js

```
export function* test1() {
  yield foo(1, 2, 3)
}

export default function* test2() {
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-regenerator/babel6-expected.js

```
"use strict";

var _marked =
/*#__PURE__*/
regeneratorRuntime.mark(test1),
    _marked2 =
/*#__PURE__*/
regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/declaration-regenerator/source.js",
              lineNumber: 2,
              code: "foo(1, 2, 3)"
            }
          });

        case 2:
        case "end":
          return _context.stop();
      }
    }
  }, _marked);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-regenerator/source.js",
    lineNumber: 1,
    code: null
  }
})

function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;

        case 2:
        case "end":
          return _context2.stop();
      }
    }
  }, _marked2);
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-regenerator/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-regenerator/babel7-expected.js

```
"use strict";

var _marked = /*#__PURE__*/regeneratorRuntime.mark(test1),
    _marked2 = /*#__PURE__*/regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/declaration-regenerator/source.js",
              lineNumber: 2,
              code: "foo(1, 2, 3)"
            }
          });

        case 2:
        case "end":
          return _context.stop();
      }
    }
  }, _marked, this);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-regenerator/source.js",
    lineNumber: 1,
    code: null
  }
})
function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;
```

```
        case 2:
        case "end":
          return _context2.stop();
      }
    }
  }, _marked2, this);
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/declaration-regenerator/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/declaration-regenerator/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-basic/babel6-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-basic/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-basic/source.js",
    lineNumber: 1,
    code: null
  }
})

function* test2() {
  yield 2;
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-basic/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-basic/babel7-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-basic/source.js",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-basic/source.js",
    lineNumber: 1,
    code: null
  }
})
function* test2() {
  yield 2;
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-basic/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-basic/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
```

```
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-delegate/babel6-expected.js

```
function* test1() {
  yield* foo(1, 2, 3);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-delegate/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-delegate/babel7-expected.js

```
function* test1() {
  yield* foo(1, 2, 3);
}
Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-delegate/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-delegate/source.js

```
function* test1() {
  yield* foo(1, 2, 3)
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-expression/babel6-expected.js

```
function* test1() {
  yield Object.defineProperty(foo.bar(1, 2, 3) || {}, "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-expression/source.js",
      lineNumber: 2,
      code: "foo.bar(1, 2, 3) || {}"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-expression/source.js",
    lineNumber: 1,
    code: null
  }
})

function* test2() {
  yield 1 + 2;
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-expression/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-expression/babel7-expected.js

```
function* test1() {
  yield Object.defineProperty(foo.bar(1, 2, 3) || {}, "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-expression/source.js",
      lineNumber: 2,
      code: "foo.bar(1, 2, 3) || {}"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-expression/source.js",
    lineNumber: 1,
    code: null
  }
})
function* test2() {
  yield 1 + 2;
}
```

```
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-expression/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-expression/source.js

```
function* test1() {
  yield foo.bar(1, 2, 3) || {}
}

function* test2() {
  yield 1 + 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-method/babel6-expected.js

```
function* test1() {
  yield Object.defineProperty(foo.bar(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-method/source.js",
      lineNumber: 2,
      code: "foo.bar(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-method/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-method/babel7-expected.js

```
function* test1() {
  yield Object.defineProperty(foo.bar(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-method/source.js",
      lineNumber: 2,
      code: "foo.bar(1, 2, 3)"
    }
  });
}
Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-method/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-method/source.js

```
function* test1() {
  yield foo.bar(1, 2, 3)
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-nested/babel6-expected.js

```
function* hasNested() {
  yield Object.defineProperty(call(Object.defineProperty(function* test2() {
    yield Object.defineProperty(call(foo), "@@redux-saga/LOCATION", {
      value: {
        fileName: "test/fixtures/effect-nested/source.js",
        lineNumber: 3,
        code: "call(foo)"
      }
    });
  }, "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-nested/source.js",
      lineNumber: 2,
      code: "function* test2() {\n    yield call(foo)\n  }"
    }
  })), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-nested/source.js",
      lineNumber: 2,
      code: "call(function* test2() {\n    yield call(foo)\n  })"
    }
  });
}

Object.defineProperty(hasNested, "@@redux-saga/LOCATION", {
```

```
  value: {
    fileName: "test/fixtures/effect-nested/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-nested/babel7-expected.js

```
function* hasNested() {
  yield Object.defineProperty(call(Object.defineProperty(function* test2() {
    yield Object.defineProperty(call(foo), "@@redux-saga/LOCATION", {
      value: {
        fileName: "test/fixtures/effect-nested/source.js",
        lineNumber: 3,
        code: "call(foo)"
      }
    });
  }, "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-nested/source.js",
      lineNumber: 2,
      code: "function* test2() {\n    yield call(foo)\n  }"
    }
  })), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-nested/source.js",
      lineNumber: 2,
      code: "call(function* test2() {\n    yield call(foo)\n  })"
    }
  });
}
Object.defineProperty(hasNested, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-nested/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-nested/source.js

```
function* hasNested() {
  yield call(function* test2() {
    yield call(foo)
  })
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-object-props/babel6-expected.js

```
function* withEffectObjectProps() {
  yield Object.defineProperty(race({
    timeout: delay(3000),
    cancelled: take('CANCELLED')
  }), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/effect-object-props/source.js",
      lineNumber: 2,
      code: "race({\n    timeout: delay(3000),\n    cancelled: take('CANCELLED'),\n  })"
    }
  });
}

Object.defineProperty(withEffectObjectProps, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/effect-object-props/source.js",
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-object-props/babel7-expected.js

```
function* withEffectObjectProps() {
  yield Object.defineProperty(race({
    timeout: delay(3000),
    cancelled: take('CANCELLED')
  }), '@@redux-saga/LOCATION', {
    value: {
      fileName: 'test/fixtures/effect-object-props/source.js',
      lineNumber: 2,
      code: 'race({\n    timeout: delay(3000),\n    cancelled: take(\'CANCELLED\'),\n  })'
    }
  });
}
Object.defineProperty(withEffectObjectProps, '@@redux-saga/LOCATION', {
  value: {
    fileName: 'test/fixtures/effect-object-props/source.js',
    lineNumber: 1,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/effect-object-props/source.js

```
function* withEffectObjectProps() {
  yield race({
    timeout: delay(3000),
    cancelled: take('CANCELLED'),
  })
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/expression/babel6-expected.js

```
const saga = Object.defineProperty(function* test1() {
  yield 1;
}, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/expression/source.js",
    lineNumber: 1,
    code: "function* test1() {\n  yield 1\n}"
  }
});
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/expression/babel7-expected.js

```
const saga = Object.defineProperty(function* test1() {
  yield 1;
}, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/expression/source.js",
    lineNumber: 1,
    code: "function* test1() {\n  yield 1\n}"
  }
});
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/expression/source.js

```
const saga = function* test1() {
  yield 1
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/preset-env/babel6-expected.js

```
"use strict";

function _typeof(obj) { if (typeof Symbol === "function" && typeof Symbol.iterator === "symbol") { _typeof = function _typeof(obj) { return typeof obj;

function _toConsumableArray(arr) { return _arrayWithoutHoles(arr) || _iterableToArray(arr) || _nonIterableSpread(); }

function _nonIterableSpread() { throw new TypeError("Invalid attempt to spread non-iterable instance"); }

function _iterableToArray(iter) { if (Symbol.iterator in Object(iter) || Object.prototype.toString.call(iter) === "[object Arguments]") return Array.fro

function _arrayWithoutHoles(arr) { if (Array.isArray(arr)) { for (var i = 0, arr2 = new Array(arr.length); i < arr.length; i++) { arr2[i] = arr[i]; } re

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); }

function _defineProperties(target, props) { for (var i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumerable = descriptor.enume

function _createClass(Constructor, protoProps, staticProps) { if (protoProps) _defineProperties(Constructor.prototype, protoProps); if (staticProps) _de

function _possibleConstructorReturn(self, call) { if (call && (_typeof(call) === "object" || typeof call === "function")) { return call; } return _asser

function _assertThisInitialized(self) { if (self === void 0) { throw new ReferenceError("this hasn't been initialised - super() hasn't been called"); }

function _getPrototypeOf(o) { _getPrototypeOf = Object.setPrototypeOf ? Object.getPrototypeOf : function _getPrototypeOf(o) { return o.__proto__ || Obje

function _inherits(subClass, superClass) { if (typeof superClass !== "function" && superClass !== null) { throw new TypeError("Super expression must eit

function _setPrototypeOf(o, p) { _setPrototypeOf = Object.setPrototypeOf || function _setPrototypeOf(o, p) { o.__proto__ = p; return o; }; return _setPr

var _marked =
/*#__PURE__*/
regeneratorRuntime.mark(test1),
    _marked2 =
/*#__PURE__*/
regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/preset-env/source.js",
              lineNumber: 2,
              code: "foo(1, 2, 3)"
            }
          });

        case 2:
```

```
        case "end":
          return _context.stop();
      }
    }
  }, _marked);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/preset-env/source.js",
    lineNumber: 1,
    code: null
  }
})

function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;

        case 2:
        case "end":
          return _context2.stop();
      }
    }
  }, _marked2);
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/preset-env/source.js",
    lineNumber: 5,
    code: null
  }
})

var Component =
/*#__PURE__*/
function (_React$PureComponent) {
  _inherits(Component, _React$PureComponent);

  function Component() {
    _classCallCheck(this, Component);

    return _possibleConstructorReturn(this, _getPrototypeOf(Component).apply(this, arguments));
  }

  _createClass(Component, [{
    key: "getData",
    value:
    /*#__PURE__*/
    regeneratorRuntime.mark(function getData() {
      return regeneratorRuntime.wrap(function getData$(_context3) {
        while (1) {
          switch (_context3.prev = _context3.next) {
            case 0:
              _context3.next = 2;
              return 1;

            case 2:
            case "end":
              return _context3.stop();
          }
        }
      }, getData);
    })
  }, {
    key: "render",
    value: function render() {
      var data = _toConsumableArray(this.getData());

      return data;
    }
  }]);

  return Component;
}(React.PureComponent);
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/preset-env/babel7-expected.js

```
"use strict";

var _createClass = function () { function defineProperties(target, props) { for (var i = 0; i < props.length; i++) { var descriptor = props[i]; descript

function _toConsumableArray(arr) { if (Array.isArray(arr)) { for (var i = 0, arr2 = Array(arr.length); i < arr.length; i++) { arr2[i] = arr[i]; } return

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); }

function _possibleConstructorReturn(self, call) { if (!self) { throw new ReferenceError("this hasn't been initialised - super() hasn't been called"); }

function _inherits(subClass, superClass) { if (typeof superClass !== "function" && superClass !== null) { throw new TypeError("Super expression must eit

var _marked = /*#__PURE__*/regeneratorRuntime.mark(test1),
    _marked2 = /*#__PURE__*/regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/preset-env/source.js",
```

```
            lineNumber: 2,
            code: "foo(1, 2, 3)"
          }
        });

      case 2:
      case "end":
        return _context.stop();
    }
  }
}, _marked, this);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/preset-env/source.js",
    lineNumber: 1,
    code: null
  }
})
function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;

        case 2:
        case "end":
          return _context2.stop();
      }
    }
  }, _marked2, this);
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/preset-env/source.js",
    lineNumber: 5,
    code: null
  }
})

var Component = function (_React$PureComponent) {
  _inherits(Component, _React$PureComponent);

  function Component() {
    _classCallCheck(this, Component);

    return _possibleConstructorReturn(this, (Component.__proto__ || Object.getPrototypeOf(Component)).apply(this, arguments));
  }

  _createClass(Component, [{
    key: "getData",
    value: /*#__PURE__*/regeneratorRuntime.mark(function getData() {
      return regeneratorRuntime.wrap(function getData$(_context3) {
        while (1) {
          switch (_context3.prev = _context3.next) {
            case 0:
              _context3.next = 2;
              return 1;

            case 2:
            case "end":
              return _context3.stop();
          }
        }
      }, getData, this);
    })
  }, {
    key: "render",
    value: function render() {
      var data = [].concat(_toConsumableArray(this.getData()));
      return data;
    }
  }]);

  return Component;
}(React.PureComponent);
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/preset-env/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
  yield 2
}

class Component extends React.PureComponent {
  *getData() {
    yield 1
  }
  render() {
    const data = [...this.getData()]
    return data
  }
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/regenerator/babel6-expected.js

```
"use strict";
```

```
var _marked =
/*#__PURE__*/
regeneratorRuntime.mark(test1),
    _marked2 =
/*#__PURE__*/
regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/regenerator/source.js",
              lineNumber: 2,
              code: "foo(1, 2, 3)"
            }
          });

        case 2:
        case "end":
          return _context.stop();
      }
    }
  }, _marked);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/regenerator/source.js",
    lineNumber: 1,
    code: null
  }
})

function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;

        case 2:
        case "end":
          return _context2.stop();
      }
    }
  }, _marked2);
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/regenerator/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/regenerator/babel7-expected.js

```
"use strict";

var _marked = /*#__PURE__*/regeneratorRuntime.mark(test1),
    _marked2 = /*#__PURE__*/regeneratorRuntime.mark(test2);

function test1() {
  return regeneratorRuntime.wrap(function test1$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          _context.next = 2;
          return Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
            value: {
              fileName: "test/fixtures/regenerator/source.js",
              lineNumber: 2,
              code: "foo(1, 2, 3)"
            }
          });

        case 2:
        case "end":
          return _context.stop();
      }
    }
  }, _marked, this);
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/regenerator/source.js",
    lineNumber: 1,
    code: null
  }
})
function test2() {
  return regeneratorRuntime.wrap(function test2$(_context2) {
    while (1) {
      switch (_context2.prev = _context2.next) {
        case 0:
          _context2.next = 2;
          return 2;

        case 2:
        case "end":
```

```
          return _context2.stop();
      }
    }
  }, _marked2, this);
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/regenerator/source.js",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/regenerator/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/typescript/babel6-expected.js

```
const sum = (a, b) => a + b;

function* tstest1() {
  const result = yield Object.defineProperty(sum(1, 2), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/typescript/source.js (source.ts)",
      lineNumber: 5,
      code: "sum(1, 2)"
    }
  });
  return result;
}

Object.defineProperty(tstest1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/typescript/source.js (source.ts)",
    lineNumber: 4,
    code: null
  }
})
const z = 1; // that's hack. since there's a problem with babel https://github.com/babel/babel/issues/7002
//# sourceMappingURL=source.js.map
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/typescript/babel7-expected.js

```
const sum = (a, b) => a + b;
function* tstest1() {
  const result = yield Object.defineProperty(sum(1, 2), "@@redux-saga/LOCATION", {
    value: {
      fileName: "test/fixtures/typescript/source.js (source.ts)",
      lineNumber: 5,
      code: "sum(1, 2)"
    }
  });
  return result;
}
Object.defineProperty(tstest1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "test/fixtures/typescript/source.js (source.ts)",
    lineNumber: 4,
    code: null
  }
})
const z = 1; // that's hack. since there's a problem with babel https://github.com/babel/babel/issues/7002
//# sourceMappingURL=source.js.map
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/typescript/source.js

```
const sum = (a, b) => a + b
function* tstest1() {
  const result = yield sum(1, 2)
  return result
}
const z = 1 // that's hack. since there's a problem with babel https://github.com/babel/babel/issues/7002
//# sourceMappingURL=source.js.map
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/typescript/source.ts

```
const sum = (a: number, b: number): number =>
    a + b;

function* tstest1(): IterableIterator<number> {
    const result = yield sum(1, 2);
    return result;
}

const z = 1; // that's hack. since there's a problem with babel https://github.com/babel/babel/issues/7002
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/use-absolute-path/babel6-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "{{absolutePath}}",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "{{absolutePath}}",
    lineNumber: 1,
    code: null
  }
})

function* test2() {
  yield 2;
}

Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "{{absolutePath}}",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/use-absolute-path/babel7-expected.js

```
function* test1() {
  yield Object.defineProperty(foo(1, 2, 3), "@@redux-saga/LOCATION", {
    value: {
      fileName: "{{absolutePath}}",
      lineNumber: 2,
      code: "foo(1, 2, 3)"
    }
  });
}

Object.defineProperty(test1, "@@redux-saga/LOCATION", {
  value: {
    fileName: "{{absolutePath}}",
    lineNumber: 1,
    code: null
  }
})
function* test2() {
  yield 2;
}
Object.defineProperty(test2, "@@redux-saga/LOCATION", {
  value: {
    fileName: "{{absolutePath}}",
    lineNumber: 5,
    code: null
  }
})
```

../redux-saga/packages/babel-plugin-redux-saga/test/fixtures/use-absolute-path/source.js

```
function* test1() {
  yield foo(1, 2, 3)
}

function* test2() {
  yield 2
}
```

../redux-saga/packages/babel-plugin-redux-saga/test/runner.test.js

```
var fs = require('fs')
var path = require('path')
var babel7 = require('@babel/core')
var babel6 = require('babel-core')

var plugin = require('babel-plugin-redux-saga')

function normalizeFilename(filename) {
  return path.normalize(filename).replace(/\\/g, '/')
}

function getExpected(expectedPath, sourcePath) {
  return fs
    .readFileSync(expectedPath, 'utf8')
    .replace(/\{\{absolutePath\}\}/g, normalizeFilename(sourcePath))
    .replace(/\r/g, '')
    .trim()
}

var testCases = [
  {
```

```
    desc: 'attach source to declaration',
    fixture: 'declaration',
  },
  {
    desc: 'attach source to export declaration',
    fixture: 'declaration-es6-modules',
  },
  {
    desc: 'attach source to export declaration when processed with regenerator',
    fixture: 'declaration-regenerator',
    options: { presets: ['env'] },
  },
  {
    desc: 'should wrap yielded call expression (no name check)',
    fixture: 'effect-basic',
  },
  {
    desc: 'should wrap method call',
    fixture: 'effect-method',
  },
  {
    desc: "shouldn't wrap delegate",
    fixture: 'effect-delegate',
  },
  {
    desc: 'should handle nested structures',
    fixture: 'effect-nested',
  },
  {
    desc: 'should handle function expression',
    fixture: 'expression',
  },
  {
    desc: 'should handle simplest expression',
    fixture: 'effect-expression',
  },
  {
    desc: 'should handle expressions in object properties',
    fixture: 'effect-object-props',
  },
  {
    desc: 'should be compatible with es2015 preset regenerator',
    fixture: 'regenerator',
    options: { presets: ['env'] },
  },
  {
    desc: 'should be compatible with env preset regenerator',
    fixture: 'preset-env',
    options: { presets: ['env'] },
  },
  {
    desc: 'should handle passed sourcemaps',
    fixture: 'typescript',
  },
  {
    desc: 'should build absolute path if useAbsolutePath option = true',
    fixture: 'use-absolute-path',
    pluginOptions: { useAbsolutePath: true },
  },
]

var testSuits = [
  {
    name: 'babel6',
    transform: babel7.transformSync,
    availablePresets: {
      env: '@babel/env',
    },
  },
  {
    name: 'babel7',
    transform: babel6.transform,
    availablePresets: {
      env: 'env',
    },
  },
]

testSuits.forEach(function (testSuit) {
  describe(testSuit.name, function () {
    testCases.forEach(function (testCase) {
      test(testCase.desc, function () {
        var sourcePath = path.join(__dirname, 'fixtures', testCase.fixture, 'source.js')
        var sourceMapPath = path.join(__dirname, 'fixtures', testCase.fixture, 'source.js.map')
        var expectedPath = path.join(__dirname, 'fixtures', testCase.fixture, testSuit.name + '-' + 'expected.js')
        var sourceCode = fs.readFileSync(sourcePath).toString()

        var inputSourceMap = fs.existsSync(sourceMapPath)
          ? JSON.parse(fs.readFileSync(sourceMapPath).toString())
          : undefined

        var options = testCase.options || {}
        var pluginOptions = testCase.pluginOptions || {}
        var presets = options.presets
          ? options.presets.map(function (p) {
              return testSuit.availablePresets[p]
            })
          : options.presets

        var actual = testSuit.transform(sourceCode, {
          compact: 'auto',
          filename: sourcePath,
          presets: presets,
          sourceMaps: Boolean(inputSourceMap),
          inputSourceMap: inputSourceMap,
          plugins: [[plugin, pluginOptions]],
        }).code

        if (fs.existsSync(expectedPath)) {
          var expected = getExpected(expectedPath, sourcePath)
          expect(actual).toBe(expected)
        } else {
```

```
        fs.writeFileSync(expectedPath, actual)
      }
    })
  })
})
```

../redux-saga/packages/core/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
        exclude: ['transform-regenerator'],
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/core/tests/channel-recipes.js

```
/* eslint-disable no-unused-vars, no-constant-condition */
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../src'
import { take, put, fork, join, call, race, cancel, actionChannel } from '../src/effects'
import { channel, buffers, END } from '../src'

test('action channel', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* saga() {
    const chan = yield actionChannel('ACTION')

    while (true) {
      const { payload } = yield take(chan)
      actual.push(payload)
      yield Promise.resolve() // block
    }
  }

  const taskP = middleware.run(saga).toPromise()

  for (var i = 0; i < 3; i++) {
    store.dispatch({
      type: 'ACTION',
      payload: i + 1,
    })
  }

  store.dispatch(END)

  return taskP.then(() => {
    // Sagas must take consecutive actions dispatched synchronously on an action channel even if it performs blocking calls
    expect(actual).toEqual([1, 2, 3])
  })
})

test('error check when constructing actionChannels', () => {
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err.message).toMatchInlineSnapshot(`"actionChannel(pattern,...): argument pattern is not valid"`)
    },
  })
  applyMiddleware(middleware)(createStore)(() => {})

  function* saga() {
    yield actionChannel(['ACTION', undefined])
  }

  const promise = middleware.run(saga).toPromise()
  return expect(promise).rejects.toThrow('argument pattern is not valid')
})

test('action channel generator', () => {
  function* saga() {
    const chan = yield actionChannel('ACTION')

    while (true) {
      const { payload } = yield take(chan)
      yield Promise.resolve() // block
    }
  }

  let gen = saga()
  let chan = actionChannel('ACTION')
  expect(gen.next().value).toEqual(chan)
  const mockChannel = channel()
  expect(gen.next(mockChannel).value).toEqual(take(mockChannel))
})
test('action channel generator with buffers', () => {
  function* saga() {
    const buffer = yield call(buffers.dropping, 1)
    const chan = yield actionChannel('ACTION', buffer)

    while (true) {
      const { payload } = yield take(chan)
```

```
      yield Promise.resolve() // block
    }
  }

  let gen = saga()
  expect(gen.next().value).toEqual(call(buffers.dropping, 1))
  let buffer = buffers.dropping(1)
  let chan = actionChannel('ACTION', buffer)
  expect(gen.next(buffer).value).toEqual(chan)
  const mockChannel = channel()
  expect(gen.next(mockChannel).value).toEqual(take(mockChannel))
})
test('channel: watcher + max workers', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* saga() {
    const chan = channel()

    try {
      for (var i = 0; i < 3; i++) {
        yield fork(worker, i + 1, chan)
      }

      while (true) {
        const { payload } = yield take('ACTION')
        yield put(chan, payload)
      }
    } finally {
      chan.close()
    }
  }

  function* worker(idx, chan) {
    let count = 0

    while (true) {
      actual.push([idx, yield take(chan)]) // 1st worker will 'sleep' after taking 2 messages on the 1st round

      if (idx === 1 && ++count === 2) {
        yield Promise.resolve()
      }
    }
  }

  const taskP = middleware.run(saga).toPromise()

  for (var i = 0; i < 10; i++) {
    store.dispatch({
      type: 'ACTION',
      payload: i + 1,
      round: 1,
    })
  }

  store.dispatch(END)

  return taskP.then(() => {
    // Saga must dispatch to free workers via channel
    expect(actual).toEqual([
      [1, 1],
      [2, 2],
      [3, 3],
      [1, 4],
      [2, 5],
      [3, 6],
      [2, 7],
      [3, 8],
      [2, 9],
      [3, 10],
    ])
  })
})
```

../redux-saga/packages/core/tests/channel.js

```
import { buffers, channel, eventChannel, END } from '../src'
import mitt from 'mitt'

const eq = (x) => (y) => x === y

test('Unbuffered channel', () => {
  let chan = channel(buffers.none())
  let actual = []

  const logger = () => (ac) => actual.push(ac)

  try {
    chan.put(undefined)
  } catch (e) {
    // channel should reject undefined messages
    expect(/provided with an undefined/.test(e.message)).toBe(true)
  }

  chan = channel(buffers.none())
  chan.take(logger(), eq(1))
  const cb = logger()
  chan.take(cb, eq(1))
  chan.put(1) // channel must notify takers

  expect(actual).toEqual([1])
  cb.cancel()
  chan.put(1) // channel must discard cancelled takes

  expect(actual).toEqual([1])
  actual = []
  chan.take(logger())
  chan.take(logger())
```

```
  chan.close() // closing a channel must resolve all takers with END

  expect(actual).toEqual([END, END])
  actual = []
  chan.take(logger()) // closed channel must resolve new takers with END

  expect(actual).toEqual([END])
  chan.put('action-after-end') // channel must reject messages after being closed

  expect(actual).toEqual([END])
})
test('buffered channel', () => {
  const buffer = []
  const spyBuffer = {
    isEmpty: () => !buffer.length,
    put: (it) => buffer.push(it),
    take: () => buffer.shift(),
  }
  let chan = channel(spyBuffer)
  let log = []

  const taker = () => {
    const _taker = (ac) => {
      _taker.called = true
      log.push(ac)
    }

    _taker.called = false
    return _taker
  }

  var t1 = taker()
  chan.take(t1) // channel must queue pending takers if there are no buffered messages

  expect([t1.called, log, buffer]).toEqual([false, [], []])
  const t2 = taker()
  chan.take(t2)
  chan.put(1) // channel must resolve the oldest pending taker with a new message

  expect([t1.called, t2.called, log, buffer]).toEqual([true, false, [1], []])
  chan.put(2)
  chan.put(3)
  chan.put(4) // channel must buffer new messages if there are no takers

  expect([buffer, t2.called, log]).toEqual([[3, 4], true, [1, 2]])
  const t3 = taker()
  chan.take(t3) // channel must resolve new takers if there are buffered messages

  expect([t3.called, buffer, log]).toEqual([true, [4], [1, 2, 3]])
  chan.close() // closing an already closed channel should be noop

  chan.close()
  chan.put('hi')
  chan.put('I said hi') // putting on an already closed channel should be noop

  expect(buffer).toEqual([4])
  chan.take(taker()) // closed channel must resolve new takers with any buffered message

  expect([log, buffer]).toEqual([[1, 2, 3, 4], []])
  chan.take(taker()) // closed channel must resolve new takers with END if there are no buffered message

  expect(log).toEqual([1, 2, 3, 4, END])
})

test('event channel', () => {
  let unsubscribeErr

  try {
    eventChannel(() => {})
  } catch (err) {
    unsubscribeErr = err
  } // eventChannel should throw if subscriber does not return a function to unsubscribe

  expect(unsubscribeErr.message).toBe('in eventChannel: subscribe should return a function to unsubscribe')
  const em = mitt()
  let chan = eventChannel((emit) => {
    em.on('*', emit)
    return () => em.off('*', emit)
  })
  let actual = []
  chan.take((ac) => actual.push(ac))
  em.emit('action-1') // eventChannel must notify takers on a new action

  expect(actual).toEqual(['action-1'])
  em.emit('action-1') // eventChannel must notify takers only once

  expect(actual).toEqual(['action-1'])
  actual = []
  chan.take(
    (ac) => actual.push(ac),
    (ac) => ac === 'action-xxx',
  )
  chan.close() // eventChannel must notify all pending takers on END

  expect(actual).toEqual([END])
  actual = []
  chan.take(
    (ac) => actual.push(ac),
    (ac) => ac === 'action-yyy',
  ) // eventChannel must notify all new takers if closed

  expect(actual).toEqual([END])
})

test('unsubscribe event channel', (done) => {
  let unsubscribed = false
  let chan = eventChannel(() => () => {
    unsubscribed = true
  })
  chan.close() // eventChannel should call unsubscribe when channel is closed

  expect(unsubscribed).toBe(true)
```

```
  unsubscribed = false
  chan = eventChannel((emitter) => {
    emitter(END)
    return () => {
      unsubscribed = true
    }
  }) // eventChannel should call unsubscribe when END event is emitted synchronously

  expect(unsubscribed).toBe(true)
  unsubscribed = false
  chan = eventChannel((emitter) => {
    setTimeout(() => emitter(END), 0)
    return () => {
      unsubscribed = true
    }
  })

  chan.take((input) => {
    // should emit END event
    expect(input).toBe(END) // eventChannel should call unsubscribe when END event is emitted asynchronously

    expect(unsubscribed).toBe(true)
    done()
  })
})

test('expanding buffer', () => {
  let chan = channel(buffers.expanding(2))
  chan.put('action-1')
  chan.put('action-2')
  chan.put('action-3')
  let actual
  chan.flush((items) => (actual = items.length))
  let expected = 3 // expanding buffer should be able to buffer more items than its initial limit

  expect(actual).toBe(expected)
})
```

../redux-saga/packages/core/tests/interpreter/all.js

```
import deferred from '@redux-saga/deferred'
import { arrayOfDeferred } from '@redux-saga/deferred'
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import { END } from '../../src'
import * as io from '../../src/effects'
test('saga parallel effects handling', () => {
  let actual
  const def = deferred()
  let cpsCb = {}

  const cps = (val, cb) =>
    (cpsCb = {
      val,
      cb,
    })

  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    actual = yield io.all([def.promise, io.cps(cps, 2), io.take('action')])
  }

  const task = middleware.run(genFn)
  Promise.resolve(1)
    .then(() => def.resolve(1))
    .then(() => cpsCb.cb(null, cpsCb.val))
    .then(() =>
      store.dispatch({
        type: 'action',
      }),
    )
  const expected = [
    1,
    2,
    {
      type: 'action',
    },
  ]
  return task.toPromise().then(() => {
    // saga must fulfill parallel effects
    expect(actual).toEqual(expected)
  })
})
test('saga empty array', () => {
  let actual
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    actual = yield io.all([])
  }

  const expected = []
  const task = middleware.run(genFn)
  return task.toPromise().then(() => {
    // saga must fulfill empty parallel effects with an empty array
    expect(actual).toEqual(expected)
  })
})
test('saga parallel effect: handling errors', () => {
  let actual
  const defs = arrayOfDeferred(2)
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  Promise.resolve(1)
    .then(() => defs[0].reject('error'))
    .then(() => defs[1].resolve(1))
```

```
  function* genFn() {
    try {
      actual = yield io.all([defs[0].promise, defs[1].promise])
    } catch (err) {
      actual = [err]
    }
  }

  const task = middleware.run(genFn)
  const expected = ['error']
  return task.toPromise().then(() => {
    // saga must catch the first error in parallel effects
    expect(actual).toEqual(expected)
  })
})
test('saga parallel effect: handling END', () => {
  let actual
  const def = deferred()
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    try {
      actual = yield io.all([def.promise, io.take('action')])
    } finally {
      actual = 'end'
    }
  }

  const task = middleware.run(genFn)
  Promise.resolve(1)
    .then(() => def.resolve(1))
    .then(() => store.dispatch(END))
  return task.toPromise().then(() => {
    // saga must end Parallel Effect if one of the effects resolve with END
    expect(actual).toEqual('end')
  })
})
test('saga parallel effect: named effects', () => {
  let actual
  const def = deferred()
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    actual = yield io.all({
      ac: io.take('action'),
      prom: def.promise,
    })
  }

  const task = middleware.run(genFn)
  Promise.resolve(1)
    .then(() => def.resolve(1))
    .then(() =>
      store.dispatch({
        type: 'action',
      }),
    )
  const expected = {
    ac: {
      type: 'action',
    },
    prom: 1,
  }
  return task.toPromise().then(() => {
    // saga must handle parallel named effects
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/base.js

```
import * as is from '@redux-saga/is'
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'

const last = (arr) => arr[arr.length - 1]

const dropRight = (n, arr) => {
  const copy = [...arr]

  while (n > 0) {
    copy.length = copy.length - 1
    n--
  }

  return copy
}

test('saga iteration', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    actual.push(yield 1)
    actual.push(yield 2)
    return 3
  }

  const task = middleware.run(genFn) // saga should return a promise of the iterator result

  expect(is.promise(task.toPromise())).toBe(true)
  return task.toPromise().then((res) => {
    // saga's iterator should return false from isRunning()
    expect(task.isRunning()).toBe(false) // saga returned promise should resolve with the iterator return value
```

```
    expect(res).toBe(3) // saga should collect yielded values from the iterator

    expect(actual).toEqual([1, 2])
  })
})
test('saga error handling', () => {
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err.message).toBe('test-error')
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function fnThrow() {
    throw new Error('test-error')
  }
  /*
    throw
  */

  function* genThrow() {
    fnThrow()
  }

  const task1 = middleware.run(genThrow)
  const promise1 = task1.toPromise().then(
    () => {
      throw new Error('saga must return a rejected promise if generator throws an uncaught error')
    },
    (
      err, // saga must return a rejected promise if generator throws an uncaught error
    ) => {
      expect(err.message).toBe('test-error')
    },
  )
  /*
    try + catch + finally
  */

  let actual = []

  function* genFinally() {
    try {
      fnThrow()
      actual.push('unreachable')
    } catch (error) {
      actual.push('caught-' + error.message)
    } finally {
      actual.push('finally')
    }
  }

  const task = middleware.run(genFinally)
  const promise2 = task.toPromise().then(() => {
    // saga must route to catch/finally blocks in the generator
    expect(actual).toEqual(['caught-test-error', 'finally'])
  })

  return Promise.all([promise1, promise2])
})
test('saga output handling', () => {
  let actual = []
  const middleware = sagaMiddleware()
  let pastStoreCreation = false

  const rootReducer = (state, action) => {
    if (pastStoreCreation) {
      actual.push(action.type)
    }

    return state
  }

  createStore(rootReducer, {}, applyMiddleware(middleware))
  pastStoreCreation = true

  function* genFn(arg) {
    yield io.put({
      type: arg,
    })
    yield io.put({
      type: 2,
    })
  }

  const task = middleware.run(genFn, 'arg')
  const expected = ['arg', 2]
  return task.toPromise().then(() => {
    // saga must handle generator output
    expect(actual).toEqual(expected)
  })
})
test('saga yielded falsy values', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    actual.push(yield false)
    actual.push(yield undefined)
    actual.push(yield null)
    actual.push(yield '')
    actual.push(yield 0)
    actual.push(yield NaN)
  }

  const task = middleware.run(genFn)
  const expected = [false, undefined, null, '', 0, NaN]
  return task.toPromise().then(() => {
    expect(isNaN(last(actual))).toBe(true) // saga must inject back yielded falsy values
```

```
    expect(dropRight(1, actual)).toEqual(dropRight(1, expected))
  })
})
```

../redux-saga/packages/core/tests/interpreter/call.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
test('saga handles call effects and resume with the resolved values', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  class C {
    constructor(val) {
      this.val = val
    }

    method() {
      return Promise.resolve(this.val)
    }
  }

  const inst1 = new C(1)
  const inst2 = new C(2)
  const inst3 = new C(3)
  const inst4 = new C(4)
  const inst5 = new C(5)
  const inst6 = new C(6)

  const eight = Symbol(8)

  function* subGen(io, arg) {
    yield Promise.resolve(null)
    return arg
  }

  function identity(arg) {
    return arg
  }

  function* genFn() {
    actual.push(yield io.call([inst1, inst1.method]))
    actual.push(yield io.call([inst2, 'method']))
    actual.push(yield io.apply(inst3, inst3.method))
    actual.push(yield io.apply(inst4, 'method'))
    actual.push(yield io.call({ context: inst5, fn: inst5.method }))
    actual.push(yield io.call({ context: inst6, fn: 'method' }))
    actual.push(yield io.call(subGen, io, 7))
    actual.push(yield io.call(identity, eight))
  }

  const task = middleware.run(genFn)
  const expected = [1, 2, 3, 4, 5, 6, 7, eight]
  return task.toPromise().then(() => {
    // saga must fulfill declarative call effects
    expect(actual).toEqual(expected)
  })
})
test('saga handles call effects and throw the rejected values inside the generator', () => {
  let actual = []
  let pastStoreCreation = false

  const rootReducer = (state, action) => {
    if (pastStoreCreation) {
      actual.push(action.type)
    }

    return {}
  }

  const middleware = sagaMiddleware()
  createStore(rootReducer, {}, applyMiddleware(middleware))
  pastStoreCreation = true

  function fail(msg) {
    return Promise.reject(msg)
  }

  function* genFnParent() {
    try {
      yield io.put({
        type: 'start',
      })
      yield io.call(fail, 'failure')
      yield io.put({
        type: 'success',
      })
    } catch (e) {
      yield io.put({
        type: e,
      })
    }
  }

  const task = middleware.run(genFnParent)
  const expected = ['start', 'failure']
  return task.toPromise().then(() => {
    // saga dispatches appropriate actions
    expect(actual).toEqual(expected)
  })
})
test("saga handles call's synchronous failures and throws in the calling generator (1)", () => {
  let actual = []
  let pastStoreCreation = false

  const rootReducer = (state, action) => {
    if (pastStoreCreation) {
```

```
    actual.push(action.type)
  }

  return {}
}

const middleware = sagaMiddleware()
createStore(rootReducer, {}, applyMiddleware(middleware))
pastStoreCreation = true

function fail(message) {
  throw new Error(message)
}

function* genFnChild() {
  try {
    yield io.put({
      type: 'startChild',
    })
    yield io.call(fail, 'child error')
    yield io.put({
      type: 'success child',
    })
  } catch (e) {
    yield io.put({
      type: 'failure child',
    })
  }
}

function* genFnParent() {
  try {
    yield io.put({
      type: 'start parent',
    })
    yield io.call(genFnChild)
    yield io.put({
      type: 'success parent',
    })
  } catch (e) {
    yield io.put({
      type: 'failure parent',
    })
  }
}

const task = middleware.run(genFnParent)
const expected = ['start parent', 'startChild', 'failure child', 'success parent']
return task.toPromise().then(() => {
  expect(actual).toEqual(expected)
})
})
test("saga handles call's synchronous failures and throws in the calling generator (2)", () => {
let actual = []
let pastStoreCreation = false

const rootReducer = (state, action) => {
  if (pastStoreCreation) {
    actual.push(action.type)
  }

  return {}
}

const middleware = sagaMiddleware()
createStore(rootReducer, {}, applyMiddleware(middleware))
pastStoreCreation = true

function fail(message) {
  throw new Error(message)
}

function* genFnChild() {
  try {
    yield io.put({
      type: 'startChild',
    })
    yield io.call(fail, 'child error')
    yield io.put({
      type: 'success child',
    })
  } catch (e) {
    yield io.put({
      type: 'failure child',
    })
    throw e
  }
}

function* genFnParent() {
  try {
    yield io.put({
      type: 'start parent',
    })
    yield io.call(genFnChild)
    yield io.put({
      type: 'success parent',
    })
  } catch (e) {
    yield io.put({
      type: 'failure parent',
    })
  }
}

const task = middleware.run(genFnParent)
const expected = ['start parent', 'startChild', 'failure child', 'failure parent']
return task.toPromise().then(() => {
  expect(actual).toEqual(expected)
})
})
test("saga handles call's synchronous failures and throws in the calling generator (2)", () => {
```

```
  let actual = []
  let pastStoreCreation = false

  const rootReducer = (state, action) => {
    if (pastStoreCreation) {
      actual.push(action.type)
    }

    return {}
  }

  const middleware = sagaMiddleware()
  createStore(rootReducer, {}, applyMiddleware(middleware))
  pastStoreCreation = true

  function* genFnChild() {
    throw 'child error'
  }

  function* genFnParent() {
    try {
      yield io.put({
        type: 'start parent',
      })
      yield io.call(genFnChild)
      yield io.put({
        type: 'success parent',
      })
    } catch (e) {
      yield io.put({
        type: e,
      })
      yield io.put({
        type: 'failure parent',
      })
    }
  }

  const task = middleware.run(genFnParent)
  const expected = ['start parent', 'child error', 'failure parent']
  return task.toPromise().then(() => {
    // saga should bubble synchronous call errors parent
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/cancellation.js

```
/* eslint-disable no-constant-condition */
import deferred from '@redux-saga/deferred'
import { arrayOfDeferred } from '@redux-saga/deferred'
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
test('saga cancellation: call effect', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let startDef = deferred()
  let cancelDef = deferred()
  let subroutineDef = deferred()
  Promise.resolve(1)
    .then(() => startDef.resolve('start'))
    .then(() => cancelDef.resolve('cancel'))
    .then(() => subroutineDef.resolve('subroutine'))

  function* main() {
    actual.push(yield startDef.promise)

    try {
      actual.push(yield io.call(subroutine))
    } finally {
      if (yield io.cancelled()) actual.push('cancelled')
    }
  }

  function* subroutine() {
    actual.push(yield 'subroutine start')

    try {
      actual.push(yield subroutineDef.promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine cancelled')
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then((v) => {
    actual.push(v)
    task.cancel()
  })
  const expected = ['start', 'subroutine start', 'cancel', 'subroutine cancelled', 'cancelled']
  return task.toPromise().then(() => {
    // cancelled call effect must throw exception inside called subroutine
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: forked children', () => {
  const actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let cancelDef = deferred()
  const rootDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const neverDef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
    .then(() => childAdef.resolve('childA resolve'))
```

```
    .then(() => rootDef.resolve('root resolve'))
    .then(() => defs[0].resolve('leaf 0 resolve'))
    .then(() => childBdef.resolve('childB resolve')) //
    .then(() => cancelDef.resolve('cancel'))
    .then(() => defs[3].resolve('leaf 3 resolve'))
    .then(() => defs[2].resolve('leaf 2 resolve'))
    .then(() => defs[1].resolve('leaf 1 resolve'))

  function* main() {
    try {
      yield io.fork(childA)
      actual.push(yield rootDef.promise)
      yield io.fork(childB)
      yield neverDef.promise
    } finally {
      if (yield io.cancelled()) actual.push('main cancelled')
    }
  }

  function* childA() {
    try {
      yield io.fork(leaf, 0)
      actual.push(yield childAdef.promise)
      yield io.fork(leaf, 1)
      yield neverDef.promise
    } finally {
      if (yield io.cancelled()) actual.push('childA cancelled')
    }
  }

  function* childB() {
    try {
      yield io.fork(leaf, 2)
      actual.push(yield childBdef.promise)
      yield io.fork(leaf, 3)
      yield neverDef.promise
    } finally {
      if (yield io.cancelled()) actual.push('childB cancelled')
    }
  }

  function* leaf(idx) {
    try {
      actual.push(yield defs[idx].promise)
    } finally {
      if (yield io.cancelled()) actual.push(`leaf ${idx} cancelled`)
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then(() => task.cancel())
  const expected = [
    'childA resolve',
    'root resolve',
    'leaf 0 resolve',
    'childB resolve',
    /* cancel */
    'main cancelled',
    'childA cancelled',
    'leaf 1 cancelled',
    'childB cancelled',
    'leaf 2 cancelled',
    'leaf 3 cancelled',
  ]
  return task.toPromise().then(() => {
    // cancelled main task must cancel all forked sub-tasks
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: take effect', () => {
  let actual = []
  let startDef = deferred()
  let cancelDef = deferred()
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* main() {
    actual.push(yield startDef.promise)

    try {
      actual.push(yield io.take('action'))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'cancelled')
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then((v) => {
    actual.push(v)
    task.cancel()
  })
  Promise.resolve(1)
    .then(() => startDef.resolve('start'))
    .then(() => cancelDef.resolve('cancel'))
    .then(() =>
      store.dispatch({
        type: 'action',
      }),
    )
  const expected = ['start', 'cancel', 'cancelled']
  return task.toPromise().then(() => {
    // cancelled take effect must stop waiting for action
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: join effect (joining from a different task)', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let cancelDef = deferred()
  let subroutineDef = deferred()
  Promise.resolve(1)
```

```
    .then(() => cancelDef.resolve('cancel'))
    .then(() => subroutineDef.resolve('subroutine'))

  function* main() {
    actual.push('start')
    let task = yield io.fork(subroutine)
    yield io.fork(callerOfJoiner1, task)
    yield io.fork(joiner2, task)
    actual.push(yield cancelDef.promise)
    yield io.cancel(task)
  }

  function* subroutine() {
    actual.push('subroutine start')

    try {
      actual.push(yield subroutineDef.promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine cancelled')
    }
  }

  function* callerOfJoiner1(task) {
    try {
      actual.push(yield io.all([io.call(joiner1, task), new Promise(() => {})]))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'caller of joiner1 cancelled')
    }
  }

  function* joiner1(task) {
    actual.push('joiner1 start')

    try {
      actual.push(yield io.join(task))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'joiner1 cancelled')
    }
  }

  function* joiner2(task) {
    actual.push('joiner2 start')

    try {
      actual.push(yield io.join(task))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'joiner2 cancelled')
    }
  }

  const task = middleware.run(main)
  /**
    Breaking change in 10.0:
  **/

  const expected = [
    'start',
    'subroutine start',
    'joiner1 start',
    'joiner2 start',
    'cancel',
    'subroutine cancelled',
    'joiner1 cancelled',
    'caller of joiner1 cancelled',
    'joiner2 cancelled',
  ]
  return task.toPromise().then(() => {
    // cancelled task must cancel foreing joiners
    expect(actual).toEqual(expected)
  })
})
test("saga cancellation: join effect (join from the same task's parent)", () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let startDef = deferred()
  let cancelDef = deferred()
  let subroutineDef = deferred()
  Promise.resolve(1)
    .then(() => startDef.resolve('start'))
    .then(() => cancelDef.resolve('cancel'))
    .then(() => subroutineDef.resolve('subroutine'))

  function* main() {
    actual.push(yield startDef.promise)
    let task = yield io.fork(subroutine)

    try {
      actual.push(yield io.join(task))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'cancelled')
    }
  }

  function* subroutine() {
    actual.push(yield 'subroutine start')

    try {
      actual.push(yield subroutineDef.promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine cancelled')
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then((v) => {
    actual.push(v)
    task.cancel()
  })
  /**
    Breaking change in 10.0: Since now attached forks are cancelled when their parent is cancelled
    cancellation of main will trigger in order: 1. cancel parent (main) 2. then cancel children (subroutine)
```

```
    Join cancellation has the following semantics: cancellation of a task triggers cancellation of all its
    joiners (similar to promise1.then(promise2): promise2 depends on promise1, if promise1 os cancelled,
    then so promise2 must be cancelled).
     In the present test, main is joining on of its proper children, so this would cause an endless loop, but
    since cancellation is noop on an already terminated task the deadlock wont happen
  **/

  const expected = ['start', 'subroutine start', 'cancel', 'cancelled', 'subroutine cancelled']
  return task.toPromise().then(() => {
    // cancelled routine must cancel proper joiners
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: parallel effect', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let startDef = deferred()
  let cancelDef = deferred()
  let subroutineDefs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => startDef.resolve('start'))
    .then(() => subroutineDefs[0].resolve('subroutine 1'))
    .then(() => cancelDef.resolve('cancel'))
    .then(() => subroutineDefs[1].resolve('subroutine 2'))

  function* main() {
    actual.push(yield startDef.promise)

    try {
      actual.push(yield io.all([io.call(subroutine1), io.call(subroutine2)]))
    } finally {
      if (yield io.cancelled()) actual.push(yield 'cancelled')
    }
  }

  function* subroutine1() {
    actual.push(yield 'subroutine 1 start')

    try {
      actual.push(yield subroutineDefs[0].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine 1 cancelled')
    }
  }

  function* subroutine2() {
    actual.push(yield 'subroutine 2 start')

    try {
      actual.push(yield subroutineDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine 2 cancelled')
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then((v) => {
    actual.push(v)
    task.cancel()
  })
  const expected = [
    'start',
    'subroutine 1 start',
    'subroutine 2 start',
    'subroutine 1',
    'cancel',
    'subroutine 2 cancelled',
    'cancelled',
  ]
  return task.toPromise().then(() => {
    // cancelled parallel effect must cancel all sub-effects
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: race effect', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let startDef = deferred()
  let cancelDef = deferred()
  let subroutineDefs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => startDef.resolve('start'))
    .then(() => cancelDef.resolve('cancel'))
    .then(() => subroutineDefs[0].resolve('subroutine 1'))
    .then(() => subroutineDefs[1].resolve('subroutine 2'))

  function* main() {
    actual.push(yield startDef.promise)

    try {
      actual.push(
        yield io.race({
          subroutine1: io.call(subroutine1),
          subroutine2: io.call(subroutine2),
        }),
      )
    } finally {
      if (yield io.cancelled()) actual.push(yield 'cancelled')
    }
  }

  function* subroutine1() {
    actual.push(yield 'subroutine 1 start')

    try {
      actual.push(yield subroutineDefs[0].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine cancelled')
    }
  }
```

```
  function* subroutine2() {
    actual.push(yield 'subroutine 2 start')

    try {
      actual.push(yield subroutineDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subroutine cancelled')
    }
  }

  const task = middleware.run(main)
  cancelDef.promise.then((v) => {
    actual.push(v)
    task.cancel()
  })
  const expected = [
    'start',
    'subroutine 1 start',
    'subroutine 2 start',
    'cancel',
    'subroutine cancelled',
    'subroutine cancelled',
    'cancelled',
  ]
  return task.toPromise().then(() => {
    // cancelled race effect must cancel all sub-effects
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: automatic parallel effect cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let subtask1Defs = arrayOfDeferred(2),
    subtask2Defs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => subtask1Defs[0].resolve('subtask_1'))
    .then(() => subtask2Defs[0].resolve('subtask_2'))
    .then(() => subtask1Defs[1].reject('subtask_1 rejection'))
    .then(() => subtask2Defs[1].resolve('subtask_2_2'))

  function* subtask1() {
    actual.push(yield subtask1Defs[0].promise)
    actual.push(yield subtask1Defs[1].promise)
  }

  function* subtask2() {
    try {
      actual.push(yield subtask2Defs[0].promise)
      actual.push(yield subtask2Defs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subtask 2 cancelled')
    }
  }

  function* genFn() {
    try {
      yield io.all([io.call(subtask1), io.call(subtask2)])
    } catch (e) {
      actual.push(yield `caught ${e}`)
    }
  }

  const task = middleware.run(genFn)
  const expected = ['subtask_1', 'subtask_2', 'subtask 2 cancelled', 'caught subtask_1 rejection']
  return task.toPromise().then(() => {
    // saga must cancel parallel sub-effects on rejection
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: automatic race competitor cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let winnerSubtaskDefs = arrayOfDeferred(2),
    loserSubtaskDefs = arrayOfDeferred(2),
    parallelSubtaskDefs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => winnerSubtaskDefs[0].resolve('winner_1'))
    .then(() => loserSubtaskDefs[0].resolve('loser_1'))
    .then(() => parallelSubtaskDefs[0].resolve('parallel_1'))
    .then(() => winnerSubtaskDefs[1].resolve('winner_2'))
    .then(() => loserSubtaskDefs[1].resolve('loser_2'))
    .then(() => parallelSubtaskDefs[1].resolve('parallel_2'))

  function* winnerSubtask() {
    try {
      actual.push(yield winnerSubtaskDefs[0].promise)
      actual.push(yield winnerSubtaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'winner subtask cancelled')
    }
  }

  function* loserSubtask() {
    try {
      actual.push(yield loserSubtaskDefs[0].promise)
      actual.push(yield loserSubtaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'loser subtask cancelled')
    }
  }

  function* parallelSubtask() {
    try {
      actual.push(yield parallelSubtaskDefs[0].promise)
      actual.push(yield parallelSubtaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'parallel subtask cancelled')
    }
  }
```

```
  function* genFn() {
    yield io.all([
      io.race({
        winner: io.call(winnerSubtask),
        loser: io.call(loserSubtask),
      }),
      io.call(parallelSubtask),
    ])
  }

  const task = middleware.run(genFn)
  const expected = ['winner_1', 'loser_1', 'parallel_1', 'winner_2', 'loser subtask cancelled', 'parallel_2']
  return task.toPromise().then(() => {
    // saga must cancel race competitors except for the winner
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation:  manual task cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let signIn = deferred(),
    signOut = deferred(),
    expires = arrayOfDeferred(3)
  Promise.resolve(1)
    .then(() => signIn.resolve('signIn'))
    .then(() => expires[0].resolve('expire_1'))
    .then(() => expires[1].resolve('expire_2'))
    .then(() => signOut.resolve('signOut'))
    .then(() => expires[2].resolve('expire_3'))

  function* subtask() {
    try {
      for (var i = 0; i < expires.length; i++) {
        actual.push(yield expires[i].promise)
      }
    } finally {
      if (yield io.cancelled()) actual.push(yield 'task cancelled')
    }
  }

  function* genFn() {
    actual.push(yield signIn.promise)
    const task = yield io.fork(subtask)
    actual.push(yield signOut.promise)
    yield io.cancel(task)
  }

  const task = middleware.run(genFn)
  const expected = ['signIn', 'expire_1', 'expire_2', 'signOut', 'task cancelled']
  return task.toPromise().then(() => {
    // saga must cancel forked tasks
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: nested task cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let start = deferred(),
    stop = deferred(),
    subtaskDefs = arrayOfDeferred(2),
    nestedTask1Defs = arrayOfDeferred(2),
    nestedTask2Defs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => start.resolve('start'))
    .then(() => subtaskDefs[0].resolve('subtask_1'))
    .then(() => nestedTask1Defs[0].resolve('nested_task_1_1'))
    .then(() => nestedTask2Defs[0].resolve('nested_task_2_1'))
    .then(() => stop.resolve('stop'))
    .then(() => nestedTask1Defs[1].resolve('nested_task_1_2'))
    .then(() => nestedTask2Defs[1].resolve('nested_task_2_2'))
    .then(() => subtaskDefs[1].resolve('subtask_2'))

  function* nestedTask1() {
    try {
      actual.push(yield nestedTask1Defs[0].promise)
      actual.push(yield nestedTask1Defs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'nested task 1 cancelled')
    }
  }

  function* nestedTask2() {
    try {
      actual.push(yield nestedTask2Defs[0].promise)
      actual.push(yield nestedTask2Defs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'nested task 2 cancelled')
    }
  }

  function* subtask() {
    try {
      actual.push(yield subtaskDefs[0].promise)
      yield io.all([io.call(nestedTask1), io.call(nestedTask2)])
      actual.push(yield subtaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subtask cancelled')
    }
  }

  function* genFn() {
    actual.push(yield start.promise)
    const task = yield io.fork(subtask)
    actual.push(yield stop.promise)
    yield io.cancel(task)
  }

  const task = middleware.run(genFn)
  const expected = [
```

```
    'start',
    'subtask_1',
    'nested_task_1_1',
    'nested_task_2_1',
    'stop',
    'nested task 1 cancelled',
    'nested task 2 cancelled',
    'subtask cancelled',
  ]
  return task.toPromise().then(() => {
    // saga must cancel forked task and its nested subtask
    expect(actual).toEqual(expected)
  })
})
test('saga cancellation: nested forked task cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let start = deferred(),
    stop = deferred(),
    subtaskDefs = arrayOfDeferred(2),
    nestedTaskDefs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => start.resolve('start'))
    .then(() => subtaskDefs[0].resolve('subtask_1'))
    .then(() => nestedTaskDefs[0].resolve('nested_task_1'))
    .then(() => stop.resolve('stop')) //
    .then(() => nestedTaskDefs[1].resolve('nested_task_2'))
    .then(() => subtaskDefs[1].resolve('subtask_2'))

  function* nestedTask() {
    try {
      actual.push(yield nestedTaskDefs[0].promise)
      actual.push(yield nestedTaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'nested task cancelled')
    }
  }

  function* subtask() {
    try {
      actual.push(yield subtaskDefs[0].promise)
      yield io.fork(nestedTask)
      actual.push(yield subtaskDefs[1].promise)
    } finally {
      if (yield io.cancelled()) actual.push(yield 'subtask cancelled')
    }
  }

  function* genFn() {
    actual.push(yield start.promise)
    const task = yield io.fork(subtask)
    actual.push(yield stop.promise)
    yield io.cancel(task)
  }

  const task = middleware.run(genFn)
  const expected = ['start', 'subtask_1', 'nested_task_1', 'stop', 'subtask cancelled', 'nested task cancelled']
  return task.toPromise().then(() => {
    // saga must cancel forked task and its forked nested subtask
    expect(actual).toEqual(expected)
  })
})
test('cancel should be able to cancel multiple tasks', () => {
  const defs = arrayOfDeferred(3)
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* worker(i) {
    try {
      yield defs[i].promise
    } finally {
      if (yield io.cancelled()) {
        actual.push(i)
      }
    }
  }

  function* genFn() {
    const t1 = yield io.fork(worker, 0)
    const t2 = yield io.fork(worker, 1)
    const t3 = yield io.fork(worker, 2)
    yield io.cancel([t1, t2, t3])
  }

  const task = middleware.run(genFn)
  const expected = [0, 1, 2]
  return task.toPromise().then(() => {
    // it must be possible to cancel multiple tasks at once
    expect(actual).toEqual(expected)
  })
})
test('cancel should support for self cancellation', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* worker() {
    try {
      yield io.cancel()
    } finally {
      if (yield io.cancelled()) {
        actual.push('self cancellation')
      }
    }
  }

  function* genFn() {
    yield io.fork(worker)
  }
```

```
  const task = middleware.run(genFn)
  const expected = ['self cancellation']
  return task.toPromise().then(() => {
    // it must be possible to trigger self cancellation
    expect(actual).toEqual(expected)
  })
})

test('should bubble an exception thrown during cancellation', () => {
  const expectedError = new Error('child error')
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err).toBe(expectedError)
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* child() {
    try {
      yield io.delay(1000)
    } finally {
      // eslint-disable-next-line no-unsafe-finally
      throw expectedError
    }
  }
  function* worker() {
    const taskA = yield io.fork(child)
    yield io.delay(100)
    yield io.cancel(taskA)
  }

  return expect(middleware.run(worker).toPromise()).rejects.toBe(expectedError)
})

test('task should end in cancelled state when joining cancelled child', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* child() {
    yield io.delay(0)
    yield io.cancel()
  }

  function* parent() {
    yield io.join(yield io.fork(child))
  }

  const task = middleware.run(parent)

  return task.toPromise().then(() => {
    expect(task.isCancelled()).toBe(true)
    expect(task.isRunning()).toBe(false)
    expect(task.isAborted()).toBe(false)
  })
})

test('task should end in cancelled state when parent gets cancelled', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let task

  function* child() {
    // just block
    yield new Promise(() => {})
  }

  function* parent() {
    task = yield io.fork(child)
  }

  function* worker() {
    const parentTask = yield io.fork(parent)
    yield io.delay(0)
    yield io.cancel(parentTask)
  }

  return middleware
    .run(worker)
    .toPromise()
    .then(() => {
      expect(task.isCancelled()).toBe(true)
      expect(task.isRunning()).toBe(false)
      expect(task.isAborted()).toBe(false)
    })
})
```

../redux-saga/packages/core/tests/interpreter/channel.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { buffers } from '../../src'
import * as io from '../../src/effects'
test('saga create channel for store actions', () => {
  let actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    const chan = yield io.actionChannel('action')

    for (var i = 0; i < 10; i++) {
      yield Promise.resolve(1)
      const { payload } = yield io.take(chan)
      actual.push(payload)
    }
  }

  const task = middleware.run(genFn)

  for (var i = 0; i < 10; i++) {
```

```
    store.dispatch({
      type: 'action',
      payload: i + 1,
    })
  }

  return task.toPromise().then(() => {
    // saga must queue dispatched actions
    expect(actual).toEqual([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  })
})
test('saga create channel for store actions (with buffer)', () => {
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const buffer = buffers.expanding()

  function* genFn() {
    // TODO: this might mean that we do not close / flush channels when sagas ends
    // should we clean them up automatically? or is it user's responsibility?
    let chan = yield io.actionChannel('action', buffer)
    return chan
  }

  const task = middleware.run(genFn)
  Promise.resolve().then(() => {
    for (var i = 0; i < 10; i++) {
      store.dispatch({
        type: 'action',
        payload: i + 1,
      })
    }
  })
  return task.toPromise().then(() => {
    // saga must queue dispatched actions
    expect(buffer.flush().map((item) => item.payload)).toEqual([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  })
})
```

../redux-saga/packages/core/tests/interpreter/context.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
test('saga must handle context in dynamic scoping manner', () => {
  let actual = []
  const context = {
    a: 1,
  }
  const middleware = sagaMiddleware({
    context,
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    actual.push(yield io.getContext('a'))
    yield io.setContext({
      b: 2,
    })
    yield io.fork(function* () {
      actual.push(yield io.getContext('a'))
      actual.push(yield io.getContext('b'))
      yield io.setContext({
        c: 3,
      })
      actual.push(yield io.getContext('c'))
    })
    actual.push(yield io.getContext('c'))
  }

  const task = middleware.run(genFn)
  const expected = [1, 1, 2, 3, undefined]
  return task.toPromise().then(() => {
    // saga must handle context in dynamic scoping manner
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/cps.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
test('saga cps call handling', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    try {
      yield io.cps((cb) => {
        actual.push('call 1')
        cb('err')
      })
      actual.push('call 2')
    } catch (err) {
      actual.push('call ' + err)
    }
  }

  const task = middleware.run(genFn)
  const expected = ['call 1', 'call err']
  return task.toPromise().then(() => {
    // saga must fulfill cps call effects
    expect(actual).toEqual(expected)
  })
})
test('saga synchronous cps failures handling', () => {
```

```
  let actual = []
  const middleware = sagaMiddleware()
  let pastStoreCreation = false

  const rootReducer = (state, action) => {
    if (pastStoreCreation) {
      actual.push(action.type)
    }

    return {}
  }

  createStore(rootReducer, {}, applyMiddleware(middleware))
  pastStoreCreation = true

  function* genFnChild() {
    try {
      yield io.put({
        type: 'startChild',
      })
      yield io.cps(() => {
        throw new Error('child error') //cb(null, "Ok")
      })
      yield io.put({
        type: 'success child',
      })
    } catch (e) {
      yield io.put({
        type: 'failure child',
      })
    }
  }

  function* genFnParent() {
    try {
      yield io.put({
        type: 'start parent',
      })
      yield io.call(genFnChild)
      yield io.put({
        type: 'success parent',
      })
    } catch (e) {
      yield io.put({
        type: 'failure parent',
      })
    }
  }

  const task = middleware.run(genFnParent)
  const expected = ['start parent', 'startChild', 'failure child', 'success parent']
  return task.toPromise().then(() => {
    // saga should inject call error into generator
    expect(actual).toEqual(expected)
  })
})
test('saga cps cancellation handling', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let cancelled = false

  const cpsFn = (cb) => {
    cb.cancel = () => {
      cancelled = true
    }
  }

  function* genFn() {
    const task = yield io.fork(function* () {
      yield io.cps(cpsFn)
    })
    yield io.cancel(task)
  }

  const task = middleware.run(genFn)
  return task.toPromise().then(() => {
    // saga should call cancellation function on callback
    expect(cancelled).toBe(true)
  })
})
```

../redux-saga/packages/core/tests/interpreter/effectMiddlewares.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import { call, take, all } from '../../src/effects'
test('effectMiddlewares - single', () => {
  let actual = []

  function rootReducer(state, action) {
    return action
  }

  const effectMiddleware = (next) => (effect) => {
    if (effect === apiCall) {
      Promise.resolve().then(() => next('injected value'))
      return
    }

    return next(effect)
  }

  const middleware = sagaMiddleware({
    effectMiddlewares: [effectMiddleware],
  })
  const store = createStore(rootReducer, {}, applyMiddleware(middleware))
  const apiCall = call(() => new Promise(() => {}))

  function* root() {
```

```
  actual.push(yield all([call(fnA), apiCall]))
}

function* fnA() {
  const result = []
  result.push((yield take('ACTION-1')).val)
  result.push((yield take('ACTION-2')).val)
  return result
}

const task = middleware.run(root)
Promise.resolve()
  .then(() =>
    store.dispatch({
      type: 'ACTION-1',
      val: 1,
    }),
  )
  .then(() =>
    store.dispatch({
      type: 'ACTION-2',
      val: 2,
    }),
  )
const expected = [[[1, 2], 'injected value']]
return task.toPromise().then(() => {
  // effectMiddleware must be able to intercept and resolve effect in a custom way
  expect(actual).toEqual(expected)
})
})
test('effectMiddlewares - multiple', () => {
let actual = []

function rootReducer(state, action) {
  return action
}

const effectMiddleware1 = (next) => (effect) => {
  actual.push('middleware1 received', effect)

  if (effect === apiCall1) {
    Promise.resolve().then(() => next('middleware1 injected value'))
    return
  }

  actual.push('middleware1 passed trough', effect)
  return next(effect)
}

const effectMiddleware2 = (next) => (effect) => {
  actual.push('middleware2 received', effect)

  if (effect === apiCall2) {
    Promise.resolve().then(() => next('middleware2 injected value'))
    return
  }

  actual.push('middleware2 passed trough', effect)
  return next(effect)
}

const middleware = sagaMiddleware({
  effectMiddlewares: [effectMiddleware1, effectMiddleware2],
})
createStore(rootReducer, {}, applyMiddleware(middleware))
const apiCall1 = call(() => new Promise(() => {}))
const apiCall2 = call(() => new Promise(() => {}))
const callA = call(fnA)

function* root() {
  actual.push("effect's result is", yield apiCall1)
  actual.push("effect's result is", yield callA)
  actual.push("effect's result is", yield apiCall2)
}

function* fnA() {
  return 'fnA result'
}

const task = middleware.run(root)
const expected = [
  'middleware1 received',
  apiCall1,
  'middleware2 received',
  'middleware1 injected value',
  'middleware2 passed trough',
  'middleware1 injected value',
  "effect's result is",
  'middleware1 injected value',
  'middleware1 received',
  callA,
  'middleware1 passed trough',
  callA,
  'middleware2 received',
  callA,
  'middleware2 passed trough',
  callA,
  "effect's result is",
  'fnA result',
  'middleware1 received',
  apiCall2,
  'middleware1 passed trough',
  apiCall2,
  'middleware2 received',
  apiCall2,
  "effect's result is",
  'middleware2 injected value',
]
return task.toPromise().then(() => {
  // multiple effectMiddlewares must create a chain
  expect(actual).toEqual(expected)
})
```

```
})
test('effectMiddlewares - nested task', () => {
  let actual = []

  function rootReducer(state, action) {
    return action
  }

  const effectMiddleware = (next) => (effect) => {
    if (effect === apiCall) {
      Promise.resolve().then(() => next('injected value'))
      return
    }

    return next(effect)
  }

  const middleware = sagaMiddleware({
    effectMiddlewares: [effectMiddleware],
  })
  const store = createStore(rootReducer, {}, applyMiddleware(middleware))
  const apiCall = call(() => new Promise(() => {}))

  function* root() {
    actual.push(yield call(fnA))
  }

  function* fnA() {
    actual.push((yield take('ACTION-1')).val)
    actual.push((yield take('ACTION-2')).val)
    actual.push(yield apiCall)
    return 'result'
  }

  const task = middleware.run(root)
  Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION-1',
        val: 1,
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'ACTION-2',
        val: 2,
      }),
    )
  const expected = [1, 2, 'injected value', 'result']
  return task.toPromise().then(() => {
    // effectMiddleware must be able to intercept effects from non-root sagas
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/flush.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { channel, END } from '../../src'
import * as io from '../../src/effects'
test('saga flush handling', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    const chan = yield io.call(channel)
    actual.push(yield io.flush(chan))
    yield io.put(chan, 1)
    yield io.put(chan, 2)
    yield io.put(chan, 3)
    actual.push(yield io.flush(chan))
    yield io.put(chan, 4)
    yield io.put(chan, 5)
    chan.close()
    actual.push(yield io.flush(chan))
    actual.push(yield io.flush(chan))
  }

  const task = middleware.run(genFn)
  const expected = [[], [1, 2, 3], [4, 5], END]
  return task.toPromise().then(() => {
    // saga must handle generator flushes
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/fork.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'

test('should not interpret returned effect. fork(() => effectCreator())', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const fn = () => null

  function* genFn() {
    const task = yield io.fork(() => io.call(fn))
    return task.toPromise()
  }

  return middleware
    .run(genFn)
    .toPromise()
    .then((actual) => {
```

```
      expect(actual).toEqual(io.call(fn))
    })
})

test("should not interpret returned effect. yield fork(takeEvery, 'pattern', fn)", () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const fn = () => null

  function* genFn() {
    const task = yield io.fork(io.takeEvery, 'pattern', fn)
    return task.toPromise()
  }

  return middleware
    .run(genFn)
    .toPromise()
    .then((actual) => {
      expect(actual).toEqual(io.takeEvery('pattern', fn))
    })
})

test('should interpret returned promise. fork(() => promise)', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    const task = yield io.fork(() => Promise.resolve('a'))
    return task.toPromise()
  }

  return middleware
    .run(genFn)
    .toPromise()
    .then((actual) => {
      expect(actual).toEqual('a')
    })
})

test('should handle promise that resolves undefined properly. fork(() => Promise.resolve(undefined))', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    const task = yield io.fork(() => Promise.resolve(undefined))
    return task.toPromise()
  }

  return middleware
    .run(genFn)
    .toPromise()
    .then((actual) => {
      expect(actual).toEqual(undefined)
    })
})

test('should interpret returned iterator. fork(() => iterator)', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    const task = yield io.fork(function* () {
      yield 1
      return 'b'
    })
    return task.toPromise()
  }

  return middleware
    .run(genFn)
    .toPromise()
    .then((actual) => {
      expect(actual).toEqual('b')
    })
})
```

../redux-saga/packages/core/tests/interpreter/forkJoinErrors.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { detach } from '../../src'
import * as io from '../../src/effects'
test('saga sync fork failures: functions', () => {
  let actual = []
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err).toBe('immediatelyFailingFork')
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware)) // NOTE: we'll be forking a function not a Generator

  function immediatelyFailingFork() {
    throw 'immediatelyFailingFork'
  }

  function* genParent() {
    try {
      actual.push('start parent')
      yield io.fork(immediatelyFailingFork)
      actual.push('success parent')
    } catch (e) {
      actual.push('parent caught ' + e)
    }
  }

  function* main() {
    try {
      actual.push('start main')
      yield io.call(genParent)
      actual.push('success main')
```

```
    } catch (e) {
      actual.push('main caught ' + e)
    }
  }

  const task = middleware.run(main)
  const expected = ['start main', 'start parent', 'main caught immediatelyFailingFork']
  return task.toPromise().then(() => {
    // saga should fails the parent if a forked function fails synchronously
    expect(actual).toEqual(expected)
  })
})
test('saga sync fork failures: functions/error bubbling', () => {
  let actual = []
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err.message).toMatchInlineSnapshot(`"immediatelyFailingFork"`)
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware)) // NOTE: we'll be forking a function not a Generator

  function immediatelyFailingFork() {
    throw new Error('immediatelyFailingFork')
  }

  function* genParent() {
    try {
      actual.push('start parent')
      yield io.fork(immediatelyFailingFork)
      actual.push('success parent')
    } catch (e) {
      actual.push('parent caught ' + e.message)
    }
  }

  function* main() {
    try {
      actual.push('start main')
      yield io.fork(genParent)
      actual.push('success main')
    } catch (e) {
      actual.push('main caught ' + e.message)
    }
  }

  const task = middleware.run(main)
  const expected = ['start main', 'start parent', 'uncaught immediatelyFailingFork']
  return task
    .toPromise()
    .catch((err) => {
      actual.push('uncaught ' + err.message)
    })
    .then(() => {
      // saga should propagate errors up to the root of fork tree
      expect(actual).toEqual(expected)
    })
})
test("saga fork's failures: generators", () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genChild() {
    throw 'gen error'
  }

  function* genParent() {
    try {
      actual.push('start parent')
      yield io.fork(genChild)
      actual.push('success parent')
    } catch (e) {
      actual.push('parent caught ' + e)
    }
  }

  function* main() {
    try {
      actual.push('start main')
      yield io.call(genParent)
      actual.push('success main')
    } catch (e) {
      actual.push('main caught ' + e)
    }
  }

  const task = middleware.run(main)
  const expected = ['start main', 'start parent', 'main caught gen error']
  return task.toPromise().then(() => {
    // saga should fails the parent if a forked generator fails synchronously
    expect(actual).toEqual(expected)
  })
})
test('saga sync fork failures: spawns (detached forks)', () => {
  let actual = []
  const middleware = sagaMiddleware({
    onError: (err) => {
      expect(err.message).toBe('gen error')
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genChild() {
    throw new Error('gen error')
  }

  function* main() {
    try {
      actual.push('start main')
      const task = yield io.spawn(genChild)
      actual.push('spawn ' + task.meta.name)
      actual.push('success parent')
```

```
    } catch (e) {
      actual.push('main caught ' + e.message)
    }
  }

  const task = middleware.run(main)
  const expected = ['start main', 'spawn genChild', 'success parent']
  return task.toPromise().then(() => {
    // saga should not fail a parent with errors from detached forks (using spawn)
    expect(actual).toEqual(expected)
  })
})
test('saga detached forks failures', (done) => {
  const actual = []
  const middleware = sagaMiddleware({
    onError: (err) => actual.push(err),
  })
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const ACTION_TYPE = 'ACTION_TYPE'
  const ACTION_TYPE2 = 'ACTION_TYPE2'
  const failError = new Error('fail error')

  function willFail(ac) {
    if (!ac.fail) {
      actual.push(ac.i)
      return
    }

    throw failError
  }

  const wontFail = (ac) => actual.push(ac.i)

  function* saga() {
    yield detach(io.takeEvery(ACTION_TYPE, willFail))
    yield io.takeEvery(ACTION_TYPE2, wontFail)
  }

  middleware
    .run(saga)
    .toPromise()
    .catch((err) => done.fail(err))

  const expected = [0, 1, 2, failError, 4]
  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: ACTION_TYPE,
        i: 0,
      }),
    )
    .then(() =>
      store.dispatch({
        type: ACTION_TYPE,
        i: 1,
      }),
    )
    .then(() =>
      store.dispatch({
        type: ACTION_TYPE,
        i: 2,
      }),
    )
    .then(() =>
      store.dispatch({
        type: ACTION_TYPE,
        i: 3,
        fail: true,
      }),
    )
    .then(() =>
      store.dispatch({
        type: ACTION_TYPE2,
        i: 4,
      }),
    )
    .then(() => {
      // saga should not fail a parent with errors from detached fork
      expect(actual).toEqual(expected)
      done()
    })
})
```

../redux-saga/packages/core/tests/interpreter/forkjoin.js

```
import deferred from '@redux-saga/deferred'
import * as is from '@redux-saga/is'
import { arrayOfDeferred } from '@redux-saga/deferred'
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
test('saga fork handling: generators', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  let task, task2

  function* subGen(arg) {
    yield Promise.resolve(1)
    return arg
  }

  class C {
    constructor(val) {
      this.val = val
    }

    *gen() {
      return this.val
    }
```

```
  }

  const inst = new C(2)

  function* genFn() {
    task = yield io.fork(subGen, 1)
    task2 = yield io.fork([inst, inst.gen])
  }

  const mainTask = middleware.run(genFn)
  return mainTask
    .toPromise()
    .then(() => {
      // fork result must include the name of the forked generator function
      expect(task.meta.name).toBe('subGen') // fork result must include the promise of the task result

      expect(is.promise(task.toPromise())).toBe(true)

      return Promise.all([
        // fork result must resolve with the return value of the forked task
        task.toPromise(),
        // fork must also handle generators defined as instance methods
        task2.toPromise(),
      ])
    })
    .then((res) => {
      expect(res).toEqual([1, 2])
    })
})
test('saga join handling : generators', () => {
  let actual = []
  const defs = arrayOfDeferred(2)
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* subGen(arg) {
    yield defs[1].promise // will be resolved after the action-1

    return arg
  }

  function* genFn() {
    const task = yield io.fork(subGen, 1)
    actual.push(yield defs[0].promise)
    actual.push(yield io.take('action-1'))
    actual.push(yield io.join(task))
  }

  const task = middleware.run(genFn)
  Promise.resolve(1)
    .then(() => defs[0].resolve(true))
    .then(() =>
      store.dispatch({
        type: 'action-1',
      }),
    )
    .then(() => defs[1].resolve(2)) // the result of the fork will be resolved the last
  // saga must not block and miss the 2 precedent effects

  const expected = [
    true,
    {
      type: 'action-1',
    },
    1,
  ]
  return task.toPromise().then(() => {
    // saga must not block on forked tasks, but block on joined tasks
    expect(actual).toEqual(expected)
  })
})
test('saga fork/join handling : functions', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const defs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => defs[0].resolve(true))
    .then(() => defs[1].resolve(2))

  function api() {
    return defs[1].promise
  }

  function syncFn() {
    return 'sync'
  }

  function* genFn() {
    const task = yield io.fork(api)
    const syncTask = yield io.fork(syncFn)
    actual.push(yield defs[0].promise)
    actual.push(yield io.join(task))
    actual.push(yield io.join(syncTask))
  }

  const task = middleware.run(genFn)
  const expected = [true, 2, 'sync']
  return task.toPromise().then(() => {
    // saga must not block on forked tasks, but block on joined tasks
    expect(actual).toEqual(expected)
  })
})
test('saga fork wait for attached children', () => {
  const actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const rootDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
```

```
    .then(childAdef.resolve)
    .then(rootDef.resolve)
    .then(defs[0].resolve)
    .then(childBdef.resolve)
    .then(defs[2].resolve)
    .then(defs[3].resolve)
    .then(defs[1].resolve)

  function* root() {
    yield io.fork(childA)
    yield rootDef.promise
    yield io.fork(childB)
  }

  function* childA() {
    yield io.fork(leaf, 0)
    yield childAdef.promise
    yield io.fork(leaf, 1)
  }

  function* childB() {
    yield io.fork(leaf, 2)
    yield childBdef.promise
    yield io.fork(leaf, 3)
  }

  function* leaf(idx) {
    yield defs[idx].promise
    actual.push(idx)
  }

  const task = middleware.run(root)
  return task.toPromise().then(() => {
    // parent task must wait for all forked tasks before terminating
    expect(actual).toEqual([0, 2, 3, 1])
  })
})
test('saga auto cancel forks on error', () => {
  const actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
    .then(() => childAdef.resolve('childA resolved'))
    .then(() => defs[0].resolve('leaf 1 resolved'))
    .then(() => childBdef.resolve('childB resolved'))
    .then(() => defs[1].resolve('leaf 2 resolved'))
    .then(() => mainDef.reject('main error')) //
    .then(() => defs[2].resolve('leaf 3 resolved'))
    .then(() => defs[3].resolve('leaf 4 resolved'))

  function* root() {
    try {
      actual.push(yield io.call(main))
    } catch (e) {
      actual.push('root caught ' + e)
    }
  }

  function* main() {
    try {
      yield io.fork(childA)
      yield io.fork(childB)
      actual.push(yield mainDef.promise)
    } catch (e) {
      actual.push(e)
      throw e
    } finally {
      if (yield io.cancelled()) actual.push('main cancelled')
    }
  }

  function* childA() {
    try {
      yield io.fork(leaf, 0)
      actual.push(yield childAdef.promise)
      yield io.fork(leaf, 1)
    } finally {
      if (yield io.cancelled()) actual.push('childA cancelled')
    }
  }

  function* childB() {
    try {
      yield io.fork(leaf, 2)
      yield io.fork(leaf, 3)
      actual.push(yield childBdef.promise)
    } finally {
      if (yield io.cancelled()) actual.push('childB cancelled')
    }
  }

  function* leaf(idx) {
    try {
      actual.push(yield defs[idx].promise)
    } finally {
      if (yield io.cancelled()) actual.push(`leaf ${idx + 1} cancelled`)
    }
  }

  const task = middleware.run(root)
  const expected = [
    'childA resolved',
    'leaf 1 resolved',
    'childB resolved',
    'leaf 2 resolved',
    'main error',
    'leaf 3 cancelled',
    'leaf 4 cancelled',
```

```
    'root caught main error',
  ]
  return task.toPromise().then(() => {
    // parent task must cancel all forked tasks when it aborts
    expect(actual).toEqual(expected)
  })
})
test('saga auto cancel forks on main cancelled', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const actual = []
  const rootDef = deferred()
  const mainDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
    .then(() => childAdef.resolve('childA resolved'))
    .then(() => defs[0].resolve('leaf 1 resolved'))
    .then(() => childBdef.resolve('childB resolved'))
    .then(() => defs[1].resolve('leaf 2 resolved'))
    .then(() => rootDef.resolve('root resolved'))
    .then(() => mainDef.resolve('main resolved'))
    .then(() => defs[2].resolve('leaf 3 resolved'))
    .then(() => defs[3].resolve('leaf 4 resolved'))

  function* root() {
    try {
      const task = yield io.fork(main)
      actual.push(yield rootDef.promise)
      yield io.cancel(task)
    } catch (e) {
      actual.push('root caught ' + e)
    }
  }

  function* main() {
    try {
      yield io.fork(childA)
      yield io.fork(childB)
      actual.push(yield mainDef.promise)
    } finally {
      if (yield io.cancelled()) actual.push('main cancelled')
    }
  }

  function* childA() {
    try {
      yield io.fork(leaf, 0)
      actual.push(yield childAdef.promise)
      yield io.fork(leaf, 1)
    } finally {
      if (yield io.cancelled()) actual.push('childA cancelled')
    }
  }

  function* childB() {
    try {
      yield io.fork(leaf, 2)
      yield io.fork(leaf, 3)
      actual.push(yield childBdef.promise)
    } finally {
      if (yield io.cancelled()) actual.push('childB cancelled')
    }
  }

  function* leaf(idx) {
    try {
      actual.push(yield defs[idx].promise)
    } finally {
      if (yield io.cancelled()) actual.push(`leaf ${idx + 1} cancelled`)
    }
  }

  const task = middleware.run(root)
  const expected = [
    'childA resolved',
    'leaf 1 resolved',
    'childB resolved',
    'leaf 2 resolved',
    'root resolved',
    'main cancelled',
    'leaf 3 cancelled',
    'leaf 4 cancelled',
  ]
  return task.toPromise().then(() => {
    // parent task must cancel all forked tasks when it's cancelled
    expect(actual).toEqual(expected)
  })
})
test('saga auto cancel forks if a child aborts', () => {
  const actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
    .then(() => childAdef.resolve('childA resolved'))
    .then(() => defs[0].resolve('leaf 1 resolved'))
    .then(() => childBdef.resolve('childB resolved'))
    .then(() => defs[1].resolve('leaf 2 resolved'))
    .then(() => mainDef.resolve('main resolved'))
    .then(() => defs[2].reject('leaf 3 error'))
    .then(() => defs[3].resolve('leaf 4 resolved'))

  function* root() {
    try {
      actual.push(yield io.call(main))
    } catch (e) {
      actual.push('root caught ' + e)
```

```
    }
  }

  function* main() {
    try {
      yield io.fork(childA)
      yield io.fork(childB)
      actual.push(yield mainDef.promise)
      return 'main returned'
    } finally {
      if (yield io.cancelled()) actual.push('main cancelled')
    }
  }

  function* childA() {
    try {
      yield io.fork(leaf, 0)
      actual.push(yield childAdef.promise)
      yield io.fork(leaf, 1)
    } finally {
      if (yield io.cancelled()) actual.push('childA cancelled')
    }
  }

  function* childB() {
    try {
      yield io.fork(leaf, 2)
      yield io.fork(leaf, 3)
      actual.push(yield childBdef.promise)
    } finally {
      if (yield io.cancelled()) actual.push('childB cancelled')
    }
  }

  function* leaf(idx) {
    try {
      actual.push(yield defs[idx].promise)
    } catch (e) {
      actual.push(e)
      throw e
    } finally {
      if (yield io.cancelled()) actual.push(`leaf ${idx + 1} cancelled`)
    }
  }

  const task = middleware.run(root)
  const expected = [
    'childA resolved',
    'leaf 1 resolved',
    'childB resolved',
    'leaf 2 resolved',
    'main resolved',
    'leaf 3 error',
    'leaf 4 cancelled',
    'root caught leaf 3 error',
  ]
  return task.toPromise().then(() => {
    // parent task must cancel all forked tasks when it aborts
    expect(actual).toEqual(expected)
  })
})
test('saga auto cancel parent + forks if a child aborts', () => {
  const actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainDef = deferred()
  const childAdef = deferred()
  const childBdef = deferred()
  const defs = arrayOfDeferred(4)
  Promise.resolve()
    .then(() => childAdef.resolve('childA resolved'))
    .then(() => defs[0].resolve('leaf 1 resolved'))
    .then(() => childBdef.resolve('childB resolved'))
    .then(() => defs[1].resolve('leaf 2 resolved'))
    .then(() => defs[2].reject('leaf 3 error'))
    .then(() => mainDef.resolve('main resolved'))
    .then(() => defs[3].resolve('leaf 4 resolved'))

  function* root() {
    try {
      actual.push(yield io.call(main))
    } catch (e) {
      actual.push('root caught ' + e)
    }
  }

  function* main() {
    try {
      yield io.fork(childA)
      yield io.fork(childB)
      actual.push(yield mainDef.promise)
      return 'main returned'
    } catch (e) {
      actual.push(e)
      throw e
    } finally {
      if (yield io.cancelled()) actual.push('main cancelled')
    }
  }

  function* childA() {
    try {
      yield io.fork(leaf, 0)
      actual.push(yield childAdef.promise)
      yield io.fork(leaf, 1)
    } finally {
      if (yield io.cancelled()) actual.push('childA cancelled')
    }
  }

  function* childB() {
    try {
```

```
      yield io.fork(leaf, 2)
      yield io.fork(leaf, 3)
      actual.push(yield childBdef.promise)
    } finally {
      if (yield io.cancelled()) actual.push('childB cancelled')
    }
  }

  function* leaf(idx) {
    try {
      actual.push(yield defs[idx].promise)
    } catch (e) {
      actual.push(e)
      throw e
    } finally {
      if (yield io.cancelled()) actual.push(`leaf ${idx + 1} cancelled`)
    }
  }

  const task = middleware.run(root)
  const expected = [
    'childA resolved',
    'leaf 1 resolved',
    'childB resolved',
    'leaf 2 resolved',
    'leaf 3 error',
    'leaf 4 cancelled',
    'main cancelled',
    'root caught leaf 3 error',
  ]
  return task.toPromise().then(() => {
    // parent task must cancel all forked tasks when it aborts
    expect(actual).toEqual(expected)
  })
})
test('joining multiple tasks', () => {
  const defs = arrayOfDeferred(3)
  let actual
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* worker(i) {
    return yield defs[i].promise
  }

  function* genFn() {
    const task1 = yield io.fork(worker, 0)
    const task2 = yield io.fork(worker, 1)
    const task3 = yield io.fork(worker, 2)
    actual = yield io.join([task1, task2, task3])
  }

  const mainTask = middleware.run(genFn)
  Promise.resolve()
    .then(() => defs[0].resolve(1))
    .then(() => defs[2].resolve(3))
    .then(() => defs[1].resolve(2))
  const expected = [1, 2, 3]
  return mainTask.toPromise().then(() => {
    // it must be possible to join on multiple tasks
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/iterators.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import { arrayOfDeferred } from '@redux-saga/deferred'
import * as io from '../../src/effects'
test('saga nested iterator handling', () => {
  let actual = []
  let defs = arrayOfDeferred(3)
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* child() {
    actual.push(yield defs[0].promise)
    actual.push(yield io.take('action-1'))
    actual.push(yield defs[1].promise)
    actual.push(yield io.take('action-2'))
    actual.push(yield defs[2].promise)
    actual.push(yield io.take('action-3'))
    actual.push(yield Promise.reject('child error'))
  }

  function* main() {
    try {
      yield child()
    } catch (e) {
      actual.push('caught ' + e)
    }
  }

  const expected = [
    1,
    {
      type: 'action-1',
    },
    2,
    {
      type: 'action-2',
    },
    3,
    {
      type: 'action-3',
    },
    'caught child error',
  ]
```

```
  const task = middleware.run(main)
  Promise.resolve(1)
    .then(() => defs[0].resolve(1))
    .then(() =>
      store.dispatch({
        type: 'action-1',
      }),
    )
    .then(() => defs[1].resolve(2))
    .then(() =>
      store.dispatch({
        type: 'action-2',
      }),
    )
    .then(() => defs[2].resolve(3))
    .then(() =>
      store.dispatch({
        type: 'action-3',
      }),
    )
  return task.toPromise().then(() => {
    // saga must fulfill nested iterator effects
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/onerror.js

```
/* eslint-disable no-console */
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'

test('saga onError is optional (the default is console.error)', () => {
  let consoleError = console.error
  console.error = jest.fn()
  const restoreConsoleError = () => (console.error = consoleError)

  const expectedError = new Error('child error')
  const middleware = sagaMiddleware()

  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* child() {
    throw expectedError
  }

  function* main() {
    yield io.call(child)
  }

  const task = middleware.run(main)
  return task
    .toPromise()
    .catch((err) => {
      // saga does not blow up without onError
      expect(err).toBe(expectedError)
      expect(console.error.mock.calls).toMatchInlineSnapshot(`
Array [
  Array [
    [Error: child error],
  ],
  Array [
    "The above error occurred in task child
    created by main
",
  ],
]
`)
    })
    .then(restoreConsoleError, restoreConsoleError)
})

test('saga onError is called for uncaught error (thrown Error instance)', () => {
  const middleware = sagaMiddleware({
    onError: (err) => {
      actual = err
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const expectedError = new Error('child error')
  let actual

  function* child() {
    throw expectedError
  }

  function* main() {
    yield io.call(child)
  }

  const task = middleware.run(main)
  return task.toPromise().catch(() => {
    // saga passes thrown Error instance in onError handler
    expect(actual).toBe(expectedError)
  })
})

test('saga onError is called for uncaught error (thrown primitive)', () => {
  const middleware = sagaMiddleware({
    onError: (err) => {
      actual = err
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))
  const expectedError = new Error('child error')
  let actual

  function* child() {
```

```
    throw expectedError
  }

  function* main() {
    yield io.call(child)
  }

  const task = middleware.run(main)
  return task.toPromise().catch(() => {
    // saga passes thrown primitive in onError handler
    expect(actual).toBe(expectedError)
  })
})

test('saga onError is not called for caught errors', () => {
  const expectedError = new Error('child error')
  let actual
  let caught
  const middleware = sagaMiddleware({
    onError: (err) => {
      actual = err
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* child() {
    throw expectedError
  }

  function* main() {
    try {
      yield io.call(child)
    } catch (err) {
      caught = err
    }
  }

  const task = middleware.run(main)
  return task.toPromise().then(() => {
    // saga must not call onError
    expect(actual).toBe(undefined) // parent must catch error

    expect(caught).toBe(expectedError)
  })
})
```

../redux-saga/packages/core/tests/interpreter/promise.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
test('saga native promise handling', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    try {
      actual.push(yield Promise.resolve(1))
      actual.push(yield Promise.reject('error'))
    } catch (e) {
      actual.push('caught ' + e)
    }
  }

  const task = middleware.run(genFn)
  return task.toPromise().then(() => {
    // saga should handle promise resolved/rejected values
    expect(actual).toEqual([1, 'caught error'])
  })
})
test('saga native promise handling: undefined errors', () => {
  let actual = []
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* genFn() {
    try {
      actual.push(yield Promise.reject())
    } catch (e) {
      actual.push('caught ' + e)
    }
  }

  const task = middleware.run(genFn)
  return task.toPromise().then(() => {
    // saga should throw if Promise rejected with an undefined error
    expect(actual).toEqual(['caught undefined'])
  })
})
```

../redux-saga/packages/core/tests/interpreter/put.js

```
import { createStore, applyMiddleware } from 'redux'
import * as io from '../../src/effects'
import deferred from '@redux-saga/deferred'
import createSagaMiddleware, { channel, END, stdChannel } from '../../src'

const thunk = () => (next) => (action) => {
  if (typeof action.then === 'function') {
    return action
  }

  next(action)
}

test('saga put handling', () => {
  let actual = []
```

```
  const spy = () => (next) => (action) => {
    actual.push(action.type)
    next(action)
  }

  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(spy, sagaMiddleware)(createStore)(() => {})

  function* genFn(arg) {
    yield io.put({
      type: arg,
    })
    yield io.put({
      type: 2,
    })
  }

  const task = sagaMiddleware.run(genFn, 'arg')
  const expected = ['arg', 2]
  return task.toPromise().then(() => {
    // saga must handle generator puts
    expect(actual).toEqual(expected)
  })
})

test('saga put in a channel', () => {
  const buffer = []
  const spyBuffer = {
    isEmpty: () => !buffer.length,
    put: (it) => buffer.push(it),
    take: () => buffer.shift(),
  }
  const chan = channel(spyBuffer)
  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(sagaMiddleware)(createStore)(() => {})

  function* genFn(arg) {
    yield io.put(chan, arg)
    yield io.put(chan, 2)
  }

  const task = sagaMiddleware.run(genFn, 'arg')
  const expected = ['arg', 2]
  return task.toPromise().then(() => {
    // saga must handle puts on a given channel
    expect(buffer).toEqual(expected)
  })
})

test("saga async put's response handling", () => {
  let actual = []
  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(thunk, sagaMiddleware)(createStore)(() => {})

  function* genFn(arg) {
    actual.push(yield io.putResolve(Promise.resolve(arg)))
    actual.push(yield io.putResolve(Promise.resolve(2)))
  }

  const task = sagaMiddleware.run(genFn, 'arg')
  const expected = ['arg', 2]
  return task.toPromise().then(() => {
    // saga must handle async responses of generator put effects
    expect(actual).toEqual(expected)
  })
})

test("saga error put's response handling", () => {
  let actual = []
  const error = new Error('error')

  const reducer = (state, action) => {
    if (action.error) {
      throw error
    }

    return state
  }

  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(sagaMiddleware)(createStore)(reducer)

  function* genFn(arg) {
    try {
      yield io.put({
        type: arg,
        error: true,
      })
    } catch (err) {
      actual.push(err)
    }
  }

  const task = sagaMiddleware.run(genFn, 'arg')
  const expected = [error]
  return task.toPromise().then(() => {
    // saga should bubble thrown errors of generator put effects
    expect(actual).toEqual(expected)
  })
})

test("saga error putResolve's response handling", () => {
  let actual = []

  const reducer = (state) => state

  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(thunk, sagaMiddleware)(createStore)(reducer)

  function* genFn(arg) {
    try {
```

```
      actual.push(yield io.putResolve(Promise.reject(new Error('error ' + arg))))
    } catch (err) {
      actual.push(err.message)
    }
  }

  const task = sagaMiddleware.run(genFn, 'arg')
  const expected = ['error arg']
  return task.toPromise().then(() => {
    // saga must bubble thrown errors of generator putResolve effects
    expect(actual).toEqual(expected)
  })
})

test('saga nested puts handling', () => {
  let actual = []
  const sagaMiddleware = createSagaMiddleware()
  applyMiddleware(sagaMiddleware)(createStore)(() => {})

  function* genA() {
    yield io.put({
      type: 'a',
    })
    actual.push('put a')
  }

  function* genB() {
    yield io.take('a')
    yield io.put({
      type: 'b',
    })
    actual.push('put b')
  }

  function* root() {
    yield io.fork(genB) // forks genB first to be ready to take before genA starts putting

    yield io.fork(genA)
  }

  const expected = ['put a', 'put b']
  return sagaMiddleware
    .run(root)
    .toPromise()
    .then(() => {
      // saga must order nested puts by executing them after the outer puts complete
      expect(actual).toEqual(expected)
    })
})

test('puts emitted while dispatching saga need not to cause stack overflow', () => {
  function* root() {
    yield io.put({
      type: 'put a lot of actions',
    })
    yield io.delay(0)
  }

  const reducer = (state, action) => action.type

  const chan = stdChannel()
  const rawPut = chan.put
  chan.put = () => {
    for (let i = 0; i < 32768; i++) {
      rawPut({ type: 'test' })
    }
  }

  const sagaMiddleware = createSagaMiddleware({ channel: chan })
  createStore(reducer, applyMiddleware(sagaMiddleware))

  const task = sagaMiddleware.run(root)
  return task.toPromise().then(() => {
    // this saga needs to run without stack overflow
    expect(true).toBe(true)
  })
})

test('puts emitted directly after creating a task (caused by another put) should not be missed by that task', () => {
  const actual = []

  const rootReducer = (state, action) => {
    return {
      callSubscriber: action.callSubscriber,
    }
  }

  const sagaMiddleware = createSagaMiddleware()
  const store = createStore(rootReducer, undefined, applyMiddleware(sagaMiddleware))
  const saga = sagaMiddleware.run(function* () {
    yield io.take('a')
    yield io.put({
      type: 'b',
      callSubscriber: true,
    })
    yield io.take('c')
    yield io.fork(function* () {
      yield io.take('do not miss')
      actual.push("didn't get missed")
    })
  })
  store.subscribe(() => {
    if (store.getState().callSubscriber) {
      store.dispatch({
        type: 'c',
      })
      store.dispatch({
        type: 'do not miss',
      })
    }
  })
  store.dispatch({
```

```
    type: 'a',
  })
  const expected = ["didn't get missed"]
  return saga.toPromise().then(() => {
    expect(actual).toEqual(expected)
  })
})

test('END should reach tasks created after it gets dispatched', () => {
  const actual = []

  const rootReducer = () => ({})

  const sagaMiddleware = createSagaMiddleware()
  const store = createStore(rootReducer, undefined, applyMiddleware(sagaMiddleware))

  function* subTask() {
    try {
      // eslint-disable-next-line no-constant-condition
      while (true) {
        actual.push('subTask taking END')
        yield io.take('NEXT')
        actual.push('should not get here')
      }
    } finally {
      actual.push('auto ended')
    }
  }

  const def = deferred()
  const rootSaga = sagaMiddleware.run(function* () {
    // eslint-disable-next-line no-constant-condition
    while (true) {
      yield io.take('START')
      actual.push('start taken')
      yield def.promise
      actual.push('non-take effect resolved')
      yield io.fork(subTask)
      actual.push('subTask forked')
    }
  })
  Promise.resolve()
    .then(() => {
      store.dispatch({
        type: 'START',
      })
      store.dispatch(END)
    })
    .then(() => {
      def.resolve()
      store.dispatch({
        type: 'NEXT',
      })
      store.dispatch({
        type: 'START',
      })
    })
  const expected = ['start taken', 'non-take effect resolved', 'subTask taking END', 'auto ended', 'subTask forked']
  return rootSaga.toPromise().then(() => {
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/interpreter/race.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import { END } from '../../src'
import deferred from '@redux-saga/deferred'
import * as io from '../../src/effects'
test('saga race between effects handling', () => {
  let resultOfRace = 'initial'
  const timeout = deferred()
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    resultOfRace = yield io.race({
      event: io.take('action'),
      timeout: timeout.promise,
    })
  }

  const task = middleware.run(genFn)
  const expected = {
    timeout: 1,
  }
  return Promise.resolve()
    .then(() => timeout.resolve(1))
    .then(() =>
      store.dispatch({
        type: 'action',
      }),
    )
    .then(() => task.toPromise())
    .then(() => {
      // saga must fulfill race between effects
      expect(resultOfRace).toEqual(expected)
    })
})
test('saga race between array of effects handling', () => {
  let actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const timeout = deferred()

  function* genFn() {
    actual.push(yield io.race([io.take('action'), timeout.promise]))
  }
```

```
  const task = middleware.run(genFn)
  // eslint-disable-next-line no-sparse-arrays
  const expected = [[, 1]]
  return Promise.resolve()
    .then(() => timeout.resolve(1))
    .then(() =>
      store.dispatch({
        type: 'action',
      }),
    )
    .then(() => task.toPromise())
    .then(() => {
      // saga must fulfill race between array of effects
      expect(actual).toEqual(expected)
    })
})
test('saga race between effects: handle END', () => {
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const timeout = deferred()
  let resultOfRace = 'initial'
  let called = false

  function* genFn() {
    called = true
    resultOfRace = yield io.race({
      event: io.take('action'),
      task: timeout.promise,
    })
  }

  const task = middleware.run(genFn)
  return Promise.resolve()
    .then(() => store.dispatch(END))
    .then(() => timeout.resolve(1))
    .then(() => task.toPromise())
    .then(() => {
      // should run saga
      expect(called).toBe(true) // saga must end Race Effect if one of the effects resolve with END

      expect(resultOfRace).toBe('initial')
    })
})
test('saga race between sync effects', () => {
  let actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    const xChan = yield io.actionChannel('x')
    const yChan = yield io.actionChannel('y')
    yield io.take('start')
    yield io.race({
      x: io.take(xChan),
      y: io.take(yChan),
    })
    yield Promise.resolve() // waiting for next tick

    actual.push(yield io.flush(xChan), yield io.flush(yChan))
  }

  const task = middleware.run(genFn)
  const expected = [
    [],
    [
      {
        type: 'y',
      },
    ],
  ]

  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'x',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'y',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'start',
      }),
    )
    .then(() => {
      return task.toPromise()
    })
    .then(() => {
      // saga must not run effects when already completed
      expect(actual).toEqual(expected)
    })
})
test('saga race cancelling joined tasks', () => {
  const middleware = sagaMiddleware()
  applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    yield io.race({
      join: io.join([
        yield io.fork(function* () {
          yield io.delay(10)
        }),
        yield io.fork(function* () {
          yield io.delay(100)
        }),
      ]),
      timeout: io.delay(50),
    })
```

```
  }

  const task = middleware.run(genFn)

  return Promise.resolve().then(() => task.toPromise())
})
```

../redux-saga/packages/core/tests/interpreter/select.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware from '../../src'
import * as io from '../../src/effects'
import deferred from '@redux-saga/deferred'
test('saga select/getState handling', () => {
  let actual = []
  const initialState = {
    counter: 0,
    arr: [1, 2],
  }

  const counterSelector = (s) => s.counter

  const arrSelector = (s, idx) => s.arr[idx]

  const def = deferred()

  const rootReducer = (state, action) => {
    if (action.type === 'inc') {
      return {
        ...state,
        counter: state.counter + 1,
      }
    }

    return state
  }

  const middleware = sagaMiddleware()
  const store = createStore(rootReducer, initialState, applyMiddleware(middleware))

  function* genFn() {
    actual.push((yield io.select()).counter)
    actual.push(yield io.select(counterSelector))
    actual.push(yield io.select(arrSelector, 1))
    yield def.promise
    actual.push((yield io.select()).counter)
    actual.push(yield io.select(counterSelector))
  }

  const task = middleware.run(genFn)

  const expected = [0, 0, 2, 1, 1]
  return Promise.resolve()
    .then(() => {
      def.resolve()
      store.dispatch({
        type: 'inc',
      })
    })
    .then(() => {
      return task.toPromise()
    })
    .then(() => {
      // should resolve getState and select effects
      expect(actual).toEqual(expected)
    })
})
```

../redux-saga/packages/core/tests/interpreter/take.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { channel, END, eventChannel } from '../../src'
import * as io from '../../src/effects'
import mitt from 'mitt'
test('saga take from default channel', () => {
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const typeSymbol = Symbol('action-symbol')
  let actual = []

  function* genFn() {
    try {
      actual.push(yield io.take()) // take all actions

      actual.push(yield io.take('action-1')) // take only actions of type 'action-1'

      actual.push(yield io.take(['action-2', 'action-2222'])) // take either type

      actual.push(yield io.take((a) => a.isAction)) // take if match predicate

      actual.push(yield io.take(['action-3', (a) => a.isMixedWithPredicate])) // take if match any from the mixed array

      actual.push(yield io.take(['action-3', (a) => a.isMixedWithPredicate])) // take if match any from the mixed array

      actual.push(yield io.take(typeSymbol)) // take only actions of a Symbol type

      actual.push(yield io.take('never-happening-action')) //  should get END
      // TODO: never-happening-action replaced with such case is not working
      // END is not handled properly on channels?
      // const chan = channel()
      // actual.push( yield io.take(chan) ) //  should get END
    } finally {
      actual.push('auto ended')
    }
  }

  const taskP = middleware.run(genFn).toPromise()
```

```
  const expected = [
    {
      type: 'action-*',
    },
    {
      type: 'action-1',
    },
    {
      type: 'action-2',
    },
    {
      type: '',
      isAction: true,
    },
    {
      type: '',
      isMixedWithPredicate: true,
    },
    {
      type: 'action-3',
    },
    {
      type: typeSymbol,
    },
    'auto ended',
  ]
  const scenarioP = Promise.resolve(1)
    .then(() =>
      store.dispatch({
        type: 'action-*',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'action-1',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'action-2',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'unnoticeable-action',
      }),
    )
    .then(() =>
      store.dispatch({
        type: '',
        isAction: true,
      }),
    )
    .then(() =>
      store.dispatch({
        type: '',
        isMixedWithPredicate: true,
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'action-3',
      }),
    )
    .then(() =>
      store.dispatch({
        type: typeSymbol,
      }),
    )
    .then(() =>
      store.dispatch({
        ...END,
        timestamp: Date.now(),
      }),
    ) // see #316
    .then(() => {
      // saga must fulfill take Effects from default channel
      expect(actual).toEqual(expected)
    })

  return Promise.all([taskP, scenarioP])
})
test('saga take from provided channel', () => {
  const chan = channel()
  let actual = []
  const middleware = sagaMiddleware()
  applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    actual.push(yield io.takeMaybe(chan))
    actual.push(yield io.takeMaybe(chan))
    actual.push(yield io.takeMaybe(chan))
    actual.push(yield io.takeMaybe(chan))
    actual.push(yield io.takeMaybe(chan))
    actual.push(yield io.takeMaybe(chan))
  }

  const task = middleware.run(genFn)
  const expected = [1, 2, 3, 4, END, END]
  return Promise.resolve()
    .then(() => chan.put(1))
    .then(() => chan.put(2))
    .then(() => chan.put(3))
    .then(() => chan.put(4))
    .then(() => chan.close())
    .then(() => task.toPromise())
    .then(() => {
      // saga must fulfill take Effects from a provided channel
      expect(actual).toEqual(expected)
    })
})
test('saga take from eventChannel', () => {
```

```
  const em = mitt()
  const error = new Error('ERROR')
  const chan = eventChannel((emit) => {
    em.on('*', emit)
    return () => em.off('*', emit)
  })
  let actual = []
  const middleware = sagaMiddleware()
  applyMiddleware(middleware)(createStore)(() => {})

  function* genFn() {
    try {
      actual.push(yield io.take(chan))
      actual.push(yield io.take(chan))
      actual.push(yield io.take(chan))
    } catch (e) {
      actual.push('in-catch-block')
      actual.push(e)
    }
  }

  const task = middleware.run(genFn)
  const expected = ['action-1', 'action-2', 'in-catch-block', error]
  return Promise.resolve()
    .then(() => em.emit('action-1'))
    .then(() => em.emit('action-2'))
    .then(() => em.emit(error))
    .then(() => em.emit('action-after-error'))
    .then(() => task.toPromise())
    .then(() => {
      // saga must take payloads from the eventChannel, and errors from eventChannel will make the saga jump to the catch block
      expect(actual).toEqual(expected)
    })
})
```

../redux-saga/packages/core/tests/interpreter/takeSync.js

```
/* eslint-disable no-unused-vars, no-constant-condition */
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { END } from '../../src'
import { take, put, fork, join, call, all, race, cancel, takeEvery, delay } from '../../src/effects'
import { channel, buffers } from '../../src'

test('synchronous sequential takes', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* fnA() {
    actual.push(yield take('a1'))
    actual.push(yield take('a3'))
  }

  function* fnB() {
    actual.push(yield take('a2'))
  }

  function* root() {
    yield fork(fnA)
    yield fork(fnB)
  }

  store.dispatch({
    type: 'a1',
  })
  store.dispatch({
    type: 'a2',
  })
  store.dispatch({
    type: 'a3',
  })
  return Promise.resolve().then(() => {
    // Sagas must take consecutive actions dispatched synchronously
    expect(actual).toEqual([
      {
        type: 'a1',
      },
      {
        type: 'a2',
      },
      {
        type: 'a3',
      },
    ])
  })
})

test('synchronous concurrent takes', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)
  /**
    If a1 wins, then a2 cancellation means it will not take the next 'a2' action,
    dispatched immediately by the store after 'a1'; so the 2n take('a2') should take it
  **/

  function* root() {
    actual.push(
      yield race({
        a1: take('a1'),
        a2: take('a2'),
      }),
    )
    actual.push(yield take('a2'))
  }

  store.dispatch({
```

```
    type: 'a1',
  })
  store.dispatch({
    type: 'a2',
  })

  return Promise.resolve().then(() => {
    // In concurrent takes only the winner must take an action
    expect(actual).toEqual([
      {
        a1: {
          type: 'a1',
        },
      },
      {
        type: 'a2',
      },
    ])
  })
})

test('synchronous parallel takes', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    actual.push(yield all([take('a1'), take('a2')]))
  }

  store.dispatch({
    type: 'a1',
  })
  store.dispatch({
    type: 'a2',
  })

  return Promise.resolve().then(() => {
    // Saga must resolve once all parallel actions dispatched
    expect(actual).toEqual([
      [
        {
          type: 'a1',
        },
        {
          type: 'a2',
        },
      ],
    ])
  })
})

test('synchronous parallel + concurrent takes', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    actual.push(
      yield all([
        race({
          a1: take('a1'),
          a2: take('a2'),
        }),
        take('a2'),
      ]),
    )
  }

  store.dispatch({
    type: 'a1',
  })
  store.dispatch({
    type: 'a2',
  })

  return Promise.resolve().then(() => {
    // Saga must resolve once all parallel actions dispatched
    expect(actual).toEqual([
      [
        {
          a1: {
            type: 'a1',
          },
        },
        {
          type: 'a2',
        },
      ],
    ])
  })
}) //see https://github.com/reactjs/redux/issues/1240

test('startup actions', () => {
  const actual = []

  function reducer(state, action) {
    if (action.type === 'a') actual.push(action.payload)
    return true
  }

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))
  middleware.run(fnA)
  middleware.run(fnB)
  /*
    Saga starts dispatching actions immediately after being started
    But since sagas are started immediately by the saga middleware
    It means saga will dispatch actions while the store creation
    is still running (applyMiddleware has not yet returned)
```

```
  */

  function* fnB() {
    yield put({
      type: 'a',
      payload: 1,
    })
    yield put({
      type: 'a',
      payload: 2,
    })
    yield put({
      type: 'a',
      payload: 3,
    })
  }

  function* fnA() {
    actual.push('fnA-' + (yield take('a')).payload)
  }

  return Promise.resolve().then(() => {
    // Saga must be able to dispatch startup actions
    expect(actual).toEqual([1, 'fnA-1', 2, 3])
  })
})
test('synchronous takes + puts', () => {
  const actual = []

  function reducer(state, action) {
    if (action.type === 'a') actual.push(action.payload)
    return true
  }

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))
  middleware.run(root)

  function* root() {
    yield take('a')
    yield put({
      type: 'a',
      payload: 'ack-1',
    })
    yield take('a')
    yield put({
      type: 'a',
      payload: 'ack-2',
    })
  }

  store.dispatch({
    type: 'a',
    payload: 1,
  })
  store.dispatch({
    type: 'a',
    payload: 2,
  })

  return Promise.resolve().then(() => {
    // Sagas must be able to interleave takes and puts without losing actions
    expect(actual).toEqual([1, 'ack-1', 2, 'ack-2'])
  })
})

test('synchronous takes (from a channel) + puts (to the store)', () => {
  const actual = []
  const chan = channel()

  function reducer(state, action) {
    if (action.type === 'a') actual.push(action.payload)
    return true
  }

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))
  middleware.run(root)

  function* root() {
    actual.push((yield take(chan, 'a')).payload)
    yield put({
      type: 'a',
      payload: 'ack-1',
    })
    actual.push((yield take(chan, 'a')).payload)
    yield put({
      type: 'a',
      payload: 'ack-2',
    })
    yield take('never-happening-action')
  }

  chan.put({
    type: 'a',
    payload: 1,
  })
  chan.put({
    type: 'a',
    payload: 2,
  })
  chan.close()

  return Promise.resolve().then(() => {
    // Sagas must be able to interleave takes (from a channel) and puts (to the store) without losing actions
    expect(actual).toEqual([1, 'ack-1', 2, 'ack-2'])
  })
}) // see #50

test('inter-saga put/take handling', () => {
  const actual = []
  const middleware = sagaMiddleware()
```

```
  const store = createStore(() => {}, applyMiddleware(middleware))
  middleware.run(root)

  function* fnA() {
    while (true) {
      let { payload } = yield take('a')
      yield fork(someAction, payload)
    }
  }

  function* fnB() {
    yield put({
      type: 'a',
      payload: 1,
    })
    yield put({
      type: 'a',
      payload: 2,
    })
    yield put({
      type: 'a',
      payload: 3,
    })
  }

  function* someAction(payload) {
    actual.push(payload)
  }

  function* root() {
    yield all([fork(fnA), fork(fnB)])
  }

  return Promise.resolve().then(() => {
    // Sagas must take actions from each other
    expect(actual).toEqual([1, 2, 3])
  })
})

test('inter-saga put/take handling (via buffered channel)', () => {
  const actual = []
  const chan = channel()
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))

  function* fnA() {
    while (true) {
      let action = yield take(chan)
      yield call(someAction, action)
    }
  }

  function* fnB() {
    yield put(chan, 1)
    yield put(chan, 2)
    yield put(chan, 3)
    yield call(chan.close)
  }

  function* someAction(action) {
    actual.push(action)
    yield Promise.resolve()
  }

  function* root() {
    yield all([fork(fnA), fork(fnB)])
  }

  return middleware
    .run(root)
    .toPromise()
    .then(() => {
      // Sagas must take actions from each other (via buffered channel)
      expect(actual).toEqual([1, 2, 3])
    })
})
test('inter-saga send/acknowledge handling', () => {
  const actual = []

  const push = ({ type }) => actual.push(type)

  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))
  middleware.run(root)

  function* fnA() {
    push(yield take('msg-1'))
    yield put({
      type: 'ack-1',
    })
    push(yield take('msg-2'))
    yield put({
      type: 'ack-2',
    })
  }

  function* fnB() {
    yield put({
      type: 'msg-1',
    })
    push(yield take('ack-1'))
    yield put({
      type: 'msg-2',
    })
    push(yield take('ack-2'))
  }

  function* root() {
    yield all([fork(fnA), fork(fnB)])
  }

  return Promise.resolve().then(() => {
```

```
    // Sagas must take actions from each other in the right order
    expect(actual).toEqual(['msg-1', 'ack-1', 'msg-2', 'ack-2'])
  })
})
test('inter-saga send/acknowledge handling (via unbuffered channel)', () => {
  const actual = [] // non buffered channel must behave like the store

  const chan = channel(buffers.none())
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))
  middleware.run(root)

  function* fnA() {
    actual.push(yield take(chan))
    yield put(chan, 'ack-1')
    actual.push(yield take(chan))
    yield put(chan, 'ack-2')
  }

  function* fnB() {
    yield put(chan, 'msg-1')
    actual.push(yield take(chan))
    yield put(chan, 'msg-2')
    actual.push(yield take(chan))
  }

  function* root() {
    yield fork(fnA)
    yield fork(fnB)
  }

  return Promise.resolve().then(() => {
    // Sagas must take actions from each other (via unbuffered channel) in the right order
    expect(actual).toEqual(['msg-1', 'ack-1', 'msg-2', 'ack-2'])
  })
})
test('inter-saga send/acknowledge handling (via buffered channel)', () => {
  const actual = []
  const chan = channel()
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))

  function* fnA() {
    actual.push(yield take(chan))
    yield put(chan, 'ack-1')
    yield Promise.resolve()
    actual.push(yield take(chan))
    yield put(chan, 'ack-2')
  }

  function* fnB() {
    yield put(chan, 'msg-1')
    yield Promise.resolve()
    actual.push(yield take(chan))
    yield put(chan, 'msg-2')
    yield Promise.resolve()
    actual.push(yield take(chan))
  }

  function* root() {
    yield fork(fnB)
    yield fork(fnA)
  }

  return middleware
    .run(root)
    .toPromise()
    .then(() => {
      // Sagas must take actions from each other (via buffered channel) in the right order
      expect(actual).toEqual(['msg-1', 'ack-1', 'msg-2', 'ack-2'])
    })
})
test('inter-saga fork/take back from forked child 1', () => {
  const actual = []
  const chan = channel()
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))

  function* root() {
    yield all([takeEvery('TEST', takeTest1), takeEvery('TEST2', takeTest2)])
  }

  let testCounter = 0

  function* takeTest1(action) {
    if (testCounter === 0) {
      actual.push(1)
      testCounter++
      yield put({
        type: 'TEST2',
      })
    } else {
      actual.push(++testCounter)
    }
  }

  function* takeTest2(action) {
    yield all([fork(forkedPut1), fork(forkedPut2)])
  }

  function* forkedPut1() {
    yield put({
      type: 'TEST',
    })
  }

  function* forkedPut2() {
    yield put({
      type: 'TEST',
    })
  }
```

```
  const task = middleware.run(root)

  store.dispatch({
    type: 'TEST',
  })
  store.dispatch(END)

  return task.toPromise().then(() => {
    // Sagas must take actions from each forked childs doing Sync puts
    expect(actual).toEqual([1, 2, 3])
  })
})
test('deeply nested forks/puts', () => {
  const actual = []
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))

  function* s1() {
    yield fork(s2)
    actual.push(yield take('a2'))
  }

  function* s2() {
    yield fork(s3)
    actual.push(yield take('a3'))
    yield put({
      type: 'a2',
    })
  }

  function* s3() {
    yield put({
      type: 'a3',
    })
  }

  middleware.run(s1) // must schedule deeply nested forks/puts

  expect(actual).toEqual([
    {
      type: 'a3',
    },
    {
      type: 'a2',
    },
  ])
}) // #413

test('inter-saga fork/take back from forked child 2', () => {
  const actual = []
  const chan = channel()
  const middleware = sagaMiddleware()
  const store = createStore(() => {}, applyMiddleware(middleware))
  let first = true

  function* root() {
    yield takeEvery('PING', ackWorker)
  }

  function* ackWorker(action) {
    if (first) {
      first = false
      yield put({
        type: 'PING',
        val: action.val + 1,
      })
      yield take(`ACK-${action.val + 1}`)
    }

    yield put({
      type: `ACK-${action.val}`,
    })
    actual.push(1)
  }

  const task = middleware.run(root)

  store.dispatch({
    type: 'PING',
    val: 0,
  })
  store.dispatch(END)

  return task.toPromise().then(() => {
    // Sagas must take actions from each forked childs doing Sync puts
    expect(actual).toEqual([1, 1])
  })
})
test('put causing sync dispatch response in store subscriber', () => {
  const reducer = (state, action) => action.type

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))
  const actual = []
  middleware.run(root)
  store.subscribe(() => {
    if (store.getState() === 'c')
      store.dispatch({
        type: 'b',
        test: true,
      })
  })
  store.dispatch({
    type: 'a',
    test: true,
  })

  function* root() {
    // eslint-disable-next-line no-constant-condition
    while (true) {
      const { a, b } = yield race({
        a: take('a'),
```

```
        b: take('b'),
      })
      actual.push(a ? a.type : b.type)

      if (a) {
        yield put({
          type: 'c',
          test: true,
        })
        continue
      }

      yield put({
        type: 'd',
        test: true,
      })
    }
  }

  return Promise.resolve().then(() => {
    // Sagas can't miss actions dispatched by store subscribers during put handling
    expect(actual).toEqual(['a', 'b'])
  })
})

test('action dispatched in root saga should get scheduled and taken by a "sibling" take', () => {
  const reducer = (state, action) => {
    if (!state) return []

    return state.concat(action.type)
  }

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))

  function* root() {
    yield all([
      put({ type: 'FIRST' }),
      takeEvery('FIRST', function* () {
        yield put({ type: 'SECOND' })
      }),
    ])
  }

  middleware.run(root)

  return Promise.resolve().then(() => {
    expect(store.getState()).toEqual(['FIRST', 'SECOND'])
  })
})

test('action dispatched synchronously in forked task should be taken a following sync take', () => {
  const actual = []
  const reducer = (state, action) => action.type

  const middleware = sagaMiddleware()
  const store = createStore(reducer, applyMiddleware(middleware))

  function* root() {
    // force async, otherwise sync root startup prevents this from being tested appropriately
    // as the scheduler is in suspended state because of it
    yield delay(10)
    yield fork(function* () {
      yield put({ type: 'A', payload: 'foo' })
    })
    actual.push(yield take('A'))
  }

  return middleware
    .run(root)
    .toPromise()
    .then(() => {
      expect(actual).toEqual([{ type: 'A', payload: 'foo' }])
    })
})
```

../redux-saga/packages/core/tests/middleware.js

```
import { createStore, applyMiddleware } from 'redux'
import sagaMiddleware, { stdChannel } from '../src'
import * as is from '@redux-saga/is'
import { put, takeEvery } from '../src/effects'

test('middleware output', () => {
  const middleware = sagaMiddleware() // middleware factory must return a function to handle {getState, dispatch}

  expect(typeof middleware).toBe('function') // middleware returned function must take exactly 1 argument

  expect(middleware.length).toBe(1)
  const nextHandler = middleware({}) // next handler must return a function to handle action

  expect(typeof nextHandler).toBe('function') // next handler must take exactly 1 argument

  expect(nextHandler.length).toBe(1)
  const actionHandler = nextHandler() // next handler must return a function to handle action

  expect(typeof actionHandler).toBe('function') // action handler must take exactly 1 argument

  expect(actionHandler.length).toBe(1)
})

test("middleware's action handler output", () => {
  const action = {}
  const actionHandler = sagaMiddleware()({})((action) => action) // action handler must return the result of the next argument

  expect(actionHandler(action)).toBe(action)
})

test('middleware.run', () => {
```

```
  let actual

  function* saga(...args) {
    actual = args
  }

  const middleware = sagaMiddleware()

  try {
    middleware.run(function* () {})
  } catch (e) {
    // middleware.run must throw an Error when executed before the middleware is connected to a Store
    expect(e instanceof Error).toBe(true)
  }

  createStore(() => {}, applyMiddleware(middleware))
  const task = middleware.run(saga, 'argument') // middleware.run must return a Task Object

  expect(is.task(task)).toBe(true)
  const expected = ['argument'] // middleware must run the Saga and provides it with the given arguments

  expect(actual).toEqual(expected)
})

test('middleware options', () => {
  try {
    sagaMiddleware({
      onError: 42,
    })
  } catch (e) {
    // middleware factory must raise an error if `options.onError` is not a function
    expect(e.message).toBe('options.onError passed to the Saga middleware is not a function!')
  }

  const err = new Error('test')

  function* saga() {
    throw err
  }

  let actual
  const expected = err
  const options = {
    onError: (err) => (actual = err),
  }
  const middleware = sagaMiddleware(options)
  createStore(() => {}, applyMiddleware(middleware))
  middleware.run(saga) // `options.onError` is called appropriately

  expect(actual).toBe(expected)
})

test('enhance channel.put with an emitter', () => {
  const actual = []
  const channel = stdChannel()
  const rawPut = channel.put
  channel.put = (action) => {
    if (action.type === 'batch') {
      action.batch.forEach(rawPut)
      return
    }
    rawPut(action)
  }

  function* saga() {
    yield takeEvery(
      (ac) => ac.from !== 'saga',
      function* ({ type }) {
        actual.push({ saga: true, got: type })
        yield put({ type: `pong_${type}`, from: 'saga' })
      },
    )
    yield takeEvery(
      (ac) => ac.from === 'saga',
      ({ type }) => {
        actual.push({ saga: true, got: type })
      },
    )
  }

  let pastStoreCreation = false
  const rootReducer = (state, { type }) => {
    if (pastStoreCreation) {
      actual.push({ reducer: true, got: type })
    }

    return {}
  }

  const middleware = sagaMiddleware({ channel })
  const store = createStore(rootReducer, {}, applyMiddleware(middleware))
  pastStoreCreation = true

  middleware.run(saga)
  store.dispatch({ type: 'a' })
  store.dispatch({
    type: 'batch',
    batch: [{ type: 'b' }, { type: 'c' }],
  })
  store.dispatch({ type: 'd' })

  // saga must be able to take actions emitted by middleware's custom emitter
  const expected = [
    { reducer: true, got: 'a' },
    { saga: true, got: 'a' },
    { reducer: true, got: 'pong_a' },
    { saga: true, got: 'pong_a' },
    { reducer: true, got: 'batch' },
    { saga: true, got: 'b' },
    { reducer: true, got: 'pong_b' },
    { saga: true, got: 'pong_b' },
    { saga: true, got: 'c' },
```

```
    { reducer: true, got: 'pong_c' },
    { saga: true, got: 'pong_c' },
    { reducer: true, got: 'd' },
    { saga: true, got: 'd' },
    { reducer: true, got: 'pong_d' },
    { saga: true, got: 'pong_d' },
  ]
  expect(actual).toEqual(expected)
})

test('middleware.run saga arguments validation', () => {
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  expect(() => middleware.run({})).toThrow('saga argument must be a Generator function')

  expect(() => middleware.run(function* saga() {})).not.toThrow()
})
```

../redux-saga/packages/core/tests/monitoring.js

```
import { arrayOfDeferred } from '@redux-saga/deferred'
import { applyMiddleware, createStore } from 'redux'
import createSagaMiddleware, { runSaga, stdChannel } from '../src'
import * as io from '../src/effects'

function createSagaMonitor(ids, effects, actions) {
  return {
    rootSagaStarted({ effectId, saga, args }) {
      ids.push(effectId)
      effects[effectId] = { saga, args }
    },

    effectTriggered({ effectId, parentEffectId, label, effect }) {
      ids.push(effectId)
      effects[effectId] = { parentEffectId, label, effect }
    },

    effectResolved(effectId, res) {
      effects[effectId].result = res
    },

    effectRejected(effectId, err) {
      effects[effectId].error = err
    },

    effectCancelled(effectId) {
      effects[effectId].cancelled = true
    },

    actionDispatched(action) {
      actions.push(action)
    },
  }
}

test('saga middleware monitoring', async () => {
  let ids = []
  let effects = {}
  let actions = []
  const storeAction = { type: 'STORE_ACTION' }
  const sagaAction = { type: 'SAGA_ACTION' }
  const apiDefs = arrayOfDeferred(2)
  Promise.resolve(1)
    .then(() => apiDefs[0].resolve('api1'))
    .then(() => apiDefs[1].resolve('api2'))

  function api(idx) {
    return apiDefs[idx].promise
  }

  function* child() {
    yield io.call(api, 1)
    yield io.put(sagaAction)
    throw 'child error'
  }

  function* main() {
    try {
      yield io.call(api, 0)
      yield io.race({
        action: io.take('action'),
        call: io.call(child),
      })
    } catch (e) {
      void 0
    }
  }

  const sagaMonitor = createSagaMonitor(ids, effects, actions)
  const sagaMiddleware = createSagaMiddleware({ sagaMonitor })
  const store = createStore(() => ({}), applyMiddleware(sagaMiddleware))
  store.dispatch(storeAction)
  const task = sagaMiddleware.run(main)
  await task.toPromise()

  const expectedEffects = {
    [ids[0]]: { saga: main, args: [], result: task },
    [ids[1]]: { parentEffectId: ids[0], label: '', effect: io.call(api, 0), result: 'api1' },
    [ids[2]]: {
      parentEffectId: ids[0],
      label: '',
      effect: io.race({ action: io.take('action'), call: io.call(child) }),
      error: 'child error',
    },
    [ids[3]]: { parentEffectId: ids[2], label: 'action', effect: io.take('action'), cancelled: true },
    [ids[4]]: { parentEffectId: ids[2], label: 'call', effect: io.call(child), error: 'child error' },
    [ids[5]]: { parentEffectId: ids[4], label: '', effect: io.call(api, 1), result: 'api2' },
    [ids[6]]: { parentEffectId: ids[4], label: '', effect: io.put(sagaAction), result: sagaAction },
```

```
  }

  // sagaMiddleware must notify the saga monitor of Effect creation and resolution
  expect(effects).toEqual(expectedEffects)

  // sagaMiddleware must notify the saga monitor of dispatched actions
  expect(actions).toEqual([storeAction, sagaAction])
})

test('runSaga monitoring', async () => {
  let ids = []
  let effects = {}
  let actions = []
  const sagaMonitor = createSagaMonitor(ids, effects, actions)
  const channel = stdChannel()

  const dispatch = (action) => {
    sagaMonitor.actionDispatched(action)
    return action
  }

  const storeAction = { type: 'STORE_ACTION' }
  const sagaAction = { type: 'SAGA_ACTION' }

  const apiDefs = arrayOfDeferred(2)

  Promise.resolve(1)
    .then(() => apiDefs[0].resolve('api1'))
    .then(() => apiDefs[1].resolve('api2'))

  function api(idx) {
    return apiDefs[idx].promise
  }

  function* child() {
    yield io.call(api, 1)
    yield io.put(sagaAction)
    throw 'child error'
  }

  function* main() {
    try {
      yield io.call(api, 0)
      yield io.race({
        action: io.take('action'),
        call: io.call(child),
      })
    } catch (e) {
      void 0
    }
  }

  const task = runSaga(
    {
      channel,
      dispatch,
      sagaMonitor,
    },
    main,
  )
  dispatch(storeAction)

  await task.toPromise()

  const expectedEffects = {
    [ids[0]]: { saga: main, args: [], result: task },
    [ids[1]]: { parentEffectId: ids[0], label: '', effect: io.call(api, 0), result: 'api1' },
    [ids[2]]: {
      parentEffectId: ids[0],
      label: '',
      effect: io.race({ action: io.take('action'), call: io.call(child) }),
      error: 'child error',
    },
    [ids[3]]: { parentEffectId: ids[2], label: 'action', effect: io.take('action'), cancelled: true },
    [ids[4]]: { parentEffectId: ids[2], label: 'call', effect: io.call(child), error: 'child error' },
    [ids[5]]: { parentEffectId: ids[4], label: '', effect: io.call(api, 1), result: 'api2' },
    [ids[6]]: { parentEffectId: ids[4], label: '', effect: io.put(sagaAction), result: sagaAction },
  }
  // runSaga must notify the saga monitor of Effect creation and resolution
  expect(effects).toEqual(expectedEffects)

  const expectedActions = [storeAction, sagaAction]
  // runSaga must notify the saga monitor of dispatched actions
  expect(actions).toEqual(expectedActions)
})

test('saga monitors without all functions', async () => {
  const storeAction = { type: 'STORE_ACTION' }
  const sagaAction = { type: 'SAGA_ACTION' }

  const apiDefs = arrayOfDeferred(2)

  Promise.resolve(1)
    .then(() => apiDefs[0].resolve('api1'))
    .then(() => apiDefs[1].resolve('api2'))

  function api(idx) {
    return apiDefs[idx].promise
  }

  function* child() {
    yield io.call(api, 1)
    yield io.put(sagaAction)
    throw 'child error'
  }

  function* main() {
    try {
      yield io.call(api, 0)
      yield io.race({
        action: io.take('action'),
        call: io.call(child),
```

```
      })
    } catch (e) {
      void 0
    }
    return 'success'
  }

  // let's create an empty object
  const sagaMonitor = {}
  const sagaMiddleware = createSagaMiddleware({ sagaMonitor })
  const store = createStore(() => ({}), applyMiddleware(sagaMiddleware))
  store.dispatch(storeAction)
  const task = sagaMiddleware.run(main)
  // given noops to fulfill the monitor interface we have survived
  const taskResult = await task.toPromise()
  expect(taskResult).toBe('success')
})
```

../redux-saga/packages/core/tests/runSaga.js

```
import { runSaga, stdChannel } from '../src'
import { fork, take, put, select, all } from '../src/effects'

function storeLike(reducer, state) {
  const channel = stdChannel()
  return {
    channel,
    dispatch: (action) => {
      state = reducer(state, action)
      channel.put(action)
      return action
    },
    getState: () => state,
  }
}

test('runSaga', () => {
  let actual = []

  function reducer(state, action) {
    return action
  }

  const store = storeLike(reducer, {})

  const typeSelector = (a) => a.type

  const task = runSaga(store, root)

  function* root() {
    yield all([fork(fnA), fork(fnB)])
  }

  function* fnA() {
    actual.push(yield take('ACTION-1'))
    actual.push(yield select(typeSelector))
    actual.push(yield take('ACTION-2'))
    actual.push(yield select(typeSelector))
    yield put({
      type: 'ACTION-3',
    })
  }

  function* fnB() {
    actual.push(yield take('ACTION-3'))
    actual.push(yield select(typeSelector))
  }

  Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION-1',
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'ACTION-2',
      }),
    )
  const expected = [
    {
      type: 'ACTION-1',
    },
    'ACTION-1',
    {
      type: 'ACTION-2',
    },
    'ACTION-2',
    {
      type: 'ACTION-3',
    },
    'ACTION-3',
  ]
  return task.toPromise().then(() => {
    // runSaga must connect the provided iterator to the store, and run it
    expect(actual).toEqual(expected)
  })
})
```

../redux-saga/packages/core/tests/sagaHelpers/debounce.js

```
import sagaMiddleware, { channel, END } from '../../src'
import { createStore, applyMiddleware } from 'redux'
import { debounce, cancel, take } from '../../src/effects'
import delayP from '@redux-saga/delay-p'

test('debounce: sync actions', () => {
```

```
  let called = 0
  const delayMs = 33
  const largeDelayMs = delayMs + 100
  const actual = []
  const expected = [[1, 'c']]
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)

  function* saga() {
    const task = yield debounce(delayMs, 'ACTION', fnToCall)
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* fnToCall(action) {
    called++
    actual.push([called, action.payload])
  }

  return Promise.resolve()
    .then(() => {
      store.dispatch({
        type: 'ACTION',
        payload: 'a',
      })
      store.dispatch({
        type: 'ACTION',
        payload: 'b',
      })
      store.dispatch({
        type: 'ACTION',
        payload: 'c',
      })
    })
    .then(() => delayP(largeDelayMs))
    .then(() =>
      store.dispatch({
        type: 'CANCEL_WATCHER',
      }),
    )
    .then(() => {
      // should debounce sync actions and pass the lastest action to a worker
      expect(actual).toEqual(expected)
    })
})
test('debounce: async actions', () => {
  let called = 0
  const delayMs = 30
  const smallDelayMs = delayMs - 10
  const largeDelayMs = delayMs + 10
  const actual = []
  const expected = [
    [1, 'c'],
    [2, 'd'],
  ]
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)

  function* saga() {
    const task = yield debounce(delayMs, 'ACTION', fnToCall)
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* fnToCall(action) {
    called++
    actual.push([called, action.payload])
  }

  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'a',
      }),
    )
    .then(() => delayP(smallDelayMs))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'b',
      }),
    )
    .then(() => delayP(smallDelayMs))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'c',
      }),
    )
    .then(() => delayP(largeDelayMs))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'd',
      }),
    )
    .then(() => delayP(largeDelayMs))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'e',
      }),
    )
    .then(() => delayP(smallDelayMs))
    .then(() =>
      store.dispatch({
        type: 'CANCEL_WATCHER',
      }),
    )
```

```
    .then(() => {
      // should debounce async actions and pass the lastest action to a worker
      expect(actual).toEqual(expected)
    })
})
test('debounce: cancelled', () => {
  let called = 0
  const delayMs = 30
  const smallDelayMs = delayMs - 10
  const actual = []
  const expected = []
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)

  function* saga() {
    const task = yield debounce(delayMs, 'ACTION', fnToCall)
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* fnToCall(action) {
    called++
    actual.push([called, action.payload])
  }

  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 'a',
      }),
    )
    .then(() => delayP(smallDelayMs))
    .then(() =>
      store.dispatch({
        type: 'CANCEL_WATCHER',
      }),
    )
    .then(() => {
      // should not call a worker if cancelled before debounce limit is reached
      expect(actual).toEqual(expected)
    })
})
test('debounce: channel', () => {
  let called = 0
  const delayMs = 30
  const largeDelayMs = delayMs + 10
  const customChannel = channel()
  const actual = []
  const expected = [[1, 'c']]
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)

  function* saga() {
    const task = yield debounce(delayMs, customChannel, fnToCall)
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* fnToCall(dataFromChannel) {
    called++
    actual.push([called, dataFromChannel])
  }

  return Promise.resolve()
    .then(() => {
      customChannel.put('a')
      customChannel.put('b')
      customChannel.put('c')
    })
    .then(() => delayP(largeDelayMs))
    .then(() => {
      customChannel.put('d')
    })
    .then(() =>
      store.dispatch({
        type: 'CANCEL_WATCHER',
      }),
    )
    .then(() => {
      // should debounce actions from channel and pass the lastest action to a worker
      expect(actual).toEqual(expected)
    })
})
test('debounce: channel END', () => {
  let called = 0
  const delayMs = 30
  const smallDelayMs = delayMs - 10
  const customChannel = channel()
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)
  let task

  function* saga() {
    task = yield debounce(delayMs, customChannel, fnToCall)
  }

  function* fnToCall() {
    called++
  }

  return Promise.resolve()
    .then(() => delayP(smallDelayMs))
    .then(() => customChannel.put(END))
    .then(() => {
      // should finish debounce task on END
      expect(task.isRunning()).toBe(false) // should not call function if finished with END

      expect(called).toBe(0)
```

```
    })
})
test('debounce: pattern END', () => {
  let called = 0
  const delayMs = 30
  const smallDelayMs = delayMs - 10
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)
  let task

  function* saga() {
    task = yield debounce(delayMs, 'ACTION', fnToCall)
  }

  function* fnToCall() {
    called++
  }

  return Promise.resolve()
    .then(() => delayP(smallDelayMs))
    .then(() => store.dispatch(END))
    .then(() => {
      // should finish debounce task on END
      expect(task.isRunning()).toBe(false) // should not call function if finished with END

      expect(called).toBe(0)
    })
})
test('debounce: pattern END during race', () => {
  let called = 0
  const delayMs = 30
  const largeDelayMs = delayMs + 10
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)
  let task

  function* saga() {
    task = yield debounce(delayMs, 'ACTION', fnToCall)
  }

  function* fnToCall() {
    called++
  }

  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION',
      }),
    )
    .then(() => store.dispatch(END))
    .then(() => delayP(largeDelayMs))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
      }),
    )
    .then(() => delayP(largeDelayMs))
    .then(() => {
      // should interrupt race on END
      expect(called).toBe(0) // should finish debounce task on END

      expect(task.isRunning()).toBe(false)
    })
})
```

../redux-saga/packages/core/tests/sagaHelpers/delay.js

```
import sagaMiddleware from '../../src'
import { createStore, applyMiddleware } from 'redux'
import { delay, call } from '../../src/effects'
import delayP from '@redux-saga/delay-p'

test('delay', async () => {
  const actual = []
  const myVal = 'myValue'
  const expected = [true, myVal]
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))
  middleware.run(saga)

  function* saga() {
    actual.push(yield delay(1))
    actual.push(yield delay(1, myVal))
  }

  await delayP(100)

  expect(actual).toEqual(expected)
})

test('delay when the timeout value exceeds the maximum allowed value', () => {
  let actual
  const middleware = sagaMiddleware({
    onError: (err) => {
      actual = err
    },
  })
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* child() {
    yield delay(2147483648)
  }

  function* main() {
    yield call(child)
  }
```

```
  const task = middleware.run(main)
  return task.toPromise().catch(() => {
    expect(actual).toEqual(new Error('delay only supports a maximum value of 2147483647ms'))
  })
})
```

../redux-saga/packages/core/tests/sagaHelpers/retry.js

```
import sagaMiddleware from '../../src'
import { createStore, applyMiddleware } from 'redux'
import { retry } from '../../src/effects'
test('retry failing', () => {
  let called = 0
  const delayMs = 0
  const errorMessage = 'failed'
  const actual = []
  const expected = [
    ['a', 1],
    ['a', 2],
    ['a', 3],
  ]
  let error
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* saga() {
    try {
      yield retry(3, delayMs, fnToCall, 'a')
    } catch (e) {
      error = e
    }
  }

  function* fnToCall(arg1) {
    called++
    actual.push([arg1, called])
    throw new Error(errorMessage)
  }

  return middleware
    .run(saga)
    .toPromise()
    .then(() => {
      // should retry only for the defined number of times
      expect(actual).toEqual(expected) // should rethrow Error if failed more than the defined number of times

      expect(error.message).toBe(errorMessage)
    })
})
test('retry without failing', () => {
  let called = false
  const delayMs = 0
  const returnedValue = 42
  let result
  const middleware = sagaMiddleware()
  createStore(() => ({}), {}, applyMiddleware(middleware))

  function* saga() {
    result = yield retry(3, delayMs, fnToCall)
  }

  function* fnToCall() {
    if (called === false) {
      called = true
      throw new Error()
    }

    return returnedValue
  }

  return middleware
    .run(saga)
    .toPromise()
    .then(() => {
      // should return a result of called function
      expect(result).toBe(returnedValue)
    })
})
```

../redux-saga/packages/core/tests/sagaHelpers/takeEvery.js

```
import sagaMiddleware, { END } from '../../src'
import { take, cancel, takeEvery } from '../../src/effects'
import { createStore, applyMiddleware } from 'redux'
test('takeEvery', () => {
  const loop = 10
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  const mainTask = middleware.run(root)

  function* root() {
    const task = yield takeEvery('ACTION', worker, 'a1', 'a2')
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* worker(arg1, arg2, action) {
    actual.push([arg1, arg2, action.payload])
  }

  const inputTask = Promise.resolve()
    .then(() => {
      for (let i = 1; i <= loop / 2; i++)
        store.dispatch({
          type: 'ACTION',
```

```
        payload: i,
      })
  }) // the watcher should be cancelled after this
  // no further task should be forked after this
  .then(() =>
    store.dispatch({
      type: 'CANCEL_WATCHER',
    }),
  )
  .then(() => {
    for (let i = loop / 2 + 1; i <= loop; i++)
      store.dispatch({
        type: 'ACTION',
        payload: i,
      })
  })
  return Promise.all([mainTask.toPromise(), inputTask]).then(() => {
    // takeEvery must fork a worker on each action
    expect(actual).toEqual([
      ['a1', 'a2', 1],
      ['a1', 'a2', 2],
      ['a1', 'a2', 3],
      ['a1', 'a2', 4],
      ['a1', 'a2', 5],
    ])
  })
})
test('takeEvery: pattern END', () => {
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainTask = middleware.run(saga)
  let task

  function* saga() {
    task = yield takeEvery('ACTION', fnToCall)
  }

  let called = false

  function* fnToCall() {
    called = true
  }

  store.dispatch(END)
  store.dispatch({
    type: 'ACTION',
  })
  return mainTask.toPromise().then(() => {
    // should finish takeEvery task on END
    expect(task.isRunning()).toBe(false) // should not call function if finished with END

    expect(called).toBe(false)
  })
})
```

../redux-saga/packages/core/tests/sagaHelpers/takeLatest.js

```
import sagaMiddleware, { END } from '../../src'
import { createStore, applyMiddleware } from 'redux'
import { arrayOfDeferred } from '@redux-saga/deferred'
import { take, cancel, takeLatest } from '../../src/effects'
test('takeLatest', () => {
  const defs = arrayOfDeferred(4)
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    const task = yield takeLatest('ACTION', worker, 'a1', 'a2')
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* worker(arg1, arg2, action) {
    const idx = action.payload - 1
    const response = yield defs[idx].promise
    actual.push([arg1, arg2, response])
  }

  return Promise.resolve()
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 1,
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 2,
      }),
    )
    .then(() => defs[0].resolve('w-1'))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 3,
      }),
    )
    .then(() => defs[1].resolve('w-2'))
    .then(() => defs[2].resolve('w-3'))
    .then(() => {
      store.dispatch({
        type: 'ACTION',
        payload: 4,
      })
      /*
    We immediately cancel the watcher after firing the action
```

```
    The watcher should be cancelled after this
    no further task should be forked
    the last forked task should also be cancelled
    */

      store.dispatch({
        type: 'CANCEL_WATCHER',
      })
    })
    .then(() => defs[3].resolve('w-4'))
    .then(() => {
      // this one should be ignored by the watcher
      store.dispatch({
        type: 'ACTION',
        payload: 5,
      })
    })
    .then(() => {
      // takeLatest must cancel current task before forking a new task
      expect(actual).toEqual([['a1', 'a2', 'w-3']])
    })
})
test('takeLatest: pattern END', () => {
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainTask = middleware.run(saga)
  let task

  function* saga() {
    task = yield takeLatest('ACTION', fnToCall)
  }

  let called = false

  function* fnToCall() {
    called = true
  }

  store.dispatch(END)
  store.dispatch({
    type: 'ACTION',
  })
  store.dispatch({
    type: 'ACTION',
  })
  return mainTask.toPromise().then(() => {
    // should finish takeLatest task on END
    expect(task.isRunning()).toBe(false) // should not call function if finished with END

    expect(called).toBe(false)
  })
})
```

../redux-saga/packages/core/tests/sagaHelpers/takeLeading.js

```
import sagaMiddleware, { END } from '../../src'
import { createStore, applyMiddleware } from 'redux'
import { arrayOfDeferred } from '@redux-saga/deferred'
import { take, cancel, takeLeading } from '../../src/effects'
test('takeLeading', () => {
  const defs = arrayOfDeferred(4)
  const actual = []
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    const task = yield takeLeading('ACTION', worker, 'a1', 'a2')
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* worker(arg1, arg2, action) {
    const idx = action.payload - 1
    const response = yield defs[idx].promise
    actual.push([arg1, arg2, response])
  }

  return Promise.resolve(1)
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 1,
      }),
    )
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 2,
      }),
    )
    .then(() => defs[1].resolve('w-2'))
    .then(() => defs[0].resolve('w-1'))
    .then(() =>
      store.dispatch({
        type: 'ACTION',
        payload: 3,
      }),
    )
    .then(() => defs[2].resolve('w-3'))
    .then(() => {
      store.dispatch({
        type: 'ACTION',
        payload: 4,
      })
      /*
    We immediately cancel the watcher after firing the action
    The watcher should be cancelled after this
    no further task should be forked
```

```
    the last forked task should also be cancelled
    */

      store.dispatch({
        type: 'CANCEL_WATCHER',
      })
    })
    .then(() => defs[3].resolve('w-4'))
    .then(() => {
      // this one should be ignored by the watcher
      store.dispatch({
        type: 'ACTION',
        payload: 5,
      })
    })
    .then(() => {
      // takeLeading must ignore new action and keep running task until the completion
      expect(actual).toEqual([
        ['a1', 'a2', 'w-1'],
        ['a1', 'a2', 'w-3'],
      ])
    })
})
test('takeLeading: pattern END', () => {
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainTask = middleware.run(saga)
  let task

  function* saga() {
    task = yield takeLeading('ACTION', fnToCall)
  }

  let called = false

  function* fnToCall() {
    called = true
  }

  store.dispatch(END)
  store.dispatch({
    type: 'ACTION',
  })
  return mainTask.toPromise().then(() => {
    // should finish takeLeading task on END
    expect(task.isRunning()).toBe(false) // should not call function if finished with END

    expect(called).toBe(false)
  })
})
```

../redux-saga/packages/core/tests/sagaHelpers/throttle.js

```
import sagaMiddleware, { buffers, END } from '../../src'
import { createStore, applyMiddleware } from 'redux'
import delayP from '@redux-saga/delay-p'
import { take, cancel, throttle, fork } from '../../src/effects'
import { channel } from '../../src'

test('throttle', () => {
  jest.useFakeTimers()

  const actual = []
  const expected = [
    ['a1', 'a2', 0],
    ['a1', 'a2', 10],
    ['a1', 'a2', 20],
    ['a1', 'a2', 30],
    ['a1', 'a2', 34],
  ]
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    const task = yield throttle(100, 'ACTION', worker, 'a1', 'a2')
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* worker(arg1, arg2, { payload }) {
    actual.push([arg1, arg2, payload])
  }

  const dispatchedActions = []

  for (let i = 0; i < 35; i++) {
    dispatchedActions.push(
      delayP(i * 10)
        .then(() => store.dispatch({ type: 'ACTION', payload: i }))
        .then(() => jest.advanceTimersByTime(10)), // next tick
    )
  }

  Promise.resolve()
    .then(() => jest.advanceTimersByTime(1)) // just start for the smallest tick
    .then(() => jest.advanceTimersByTime(10)) // tick past first delay

  return (
    dispatchedActions[34] // wait so trailing dispatch gets processed
      .then(() => jest.advanceTimersByTime(100))
      .then(() => store.dispatch({ type: 'CANCEL_WATCHER' }))
      // shouldn't be processed cause of getting canceled
      .then(() => store.dispatch({ type: 'ACTION', payload: 40 }))
      .then(() => {
        // throttle must ignore incoming actions during throttling interval
        expect(actual).toEqual(expected)
        jest.useRealTimers()
      })
```

```
      .catch((err) => {
        jest.useRealTimers()
        throw err
      })
  )
})

test('throttle - channel', () => {
  jest.useFakeTimers()

  const actual = []
  const expected = [
    ['a1', 'a2', 0],
    ['a1', 'a2', 10],
    ['a1', 'a2', 20],
    ['a1', 'a2', 30],
    ['a1', 'a2', 34],
  ]
  const middleware = sagaMiddleware()
  const store = applyMiddleware(middleware)(createStore)(() => {})
  middleware.run(root)

  function* root() {
    const chan = channel(buffers.sliding(1))
    yield fork(listen, chan)
    yield fork(worker1, chan)
  }

  function* listen(chan) {
    while (true) {
      const action = yield take('ACTION')
      chan.put(action)
    }
  }

  function* worker1(chan) {
    const task = yield throttle(100, chan, worker2, 'a1', 'a2')
    yield take('CANCEL_WATCHER')
    yield cancel(task)
  }

  function* worker2(arg1, arg2, { payload }) {
    actual.push([arg1, arg2, payload])
  }

  const dispatchedActions = []

  for (let i = 0; i < 35; i++) {
    dispatchedActions.push(
      delayP(i * 10)
        .then(() => store.dispatch({ type: 'ACTION', payload: i }))
        .then(() => jest.advanceTimersByTime(10)), // next tick
    )
  }

  Promise.resolve()
    .then(() => jest.advanceTimersByTime(1)) // just start for the smallest tick
    .then(() => jest.advanceTimersByTime(10)) // tick past first delay

  return (
    dispatchedActions[34] // wait so trailing dispatch gets processed
      .then(() => jest.advanceTimersByTime(100))
      .then(() => store.dispatch({ type: 'CANCEL_WATCHER' }))
      // shouldn't be processed cause of getting canceled
      .then(() => store.dispatch({ type: 'ACTION', payload: 40 }))
      .then(() => {
        // throttle must ignore incoming actions during throttling interval
        expect(actual).toEqual(expected)
        jest.useRealTimers()
      })
      .catch((err) => {
        jest.useRealTimers()
        throw err
      })
  )
})

test('throttle: pattern END', () => {
  const delayMs = 20
  const middleware = sagaMiddleware()
  const store = createStore(() => ({}), {}, applyMiddleware(middleware))
  const mainTask = middleware.run(saga)
  let task

  function* saga() {
    task = yield throttle(delayMs, 'ACTION', fnToCall)
  }

  let called = false

  function* fnToCall() {
    called = true
  }

  store.dispatch(END)
  return mainTask
    .toPromise()
    .then(() =>
      store.dispatch({
        type: 'ACTION',
      }),
    )
    .then(() => delayP(2 * delayMs))
    .then(() => {
      // should finish throttle task on END
      expect(task.isRunning()).toBe(false) // should not call function if finished with END

      expect(called).toBe(false)
    })
})
```

../redux-saga/packages/core/tests/scheduler.js

```
import { asap, immediately } from '../src/internal/scheduler'

test('scheduler executes all recursively triggered tasks in order', () => {
  const actual = []
  asap(() => {
    actual.push('1')
    asap(() => {
      actual.push('2')
    })
    asap(() => {
      actual.push('3')
    })
  })
  expect(actual).toEqual(['1', '2', '3'])
})

test('scheduler when suspended queues up and executes all tasks on flush', () => {
  const actual = []
  immediately(() => {
    asap(() => {
      actual.push('1')
      asap(() => {
        actual.push('2')
      })
      asap(() => {
        actual.push('3')
      })
    })
  })
  expect(actual).toEqual(['1', '2', '3'])
})
```

../redux-saga/packages/core/tests/taskToPromise.js

```
import { TASK_CANCEL } from '@redux-saga/symbols'
import { runSaga, stdChannel } from '../src'
import { cancel, delay, fork } from '../src/effects'
import { noop } from '../src/internal/utils'

function simpleRunSaga(saga) {
  const channel = stdChannel()
  return runSaga({ channel, dispatch: channel.put, onError: noop }, saga)
}

test('calling toPromise() of an already completed task', () => {
  const result = 'result-of-saga'

  const task = simpleRunSaga(function* saga() {
    return result
  })
  expect(task.isRunning()).toBe(false)
  return expect(task.toPromise()).resolves.toBe(result)
})

test('calling toPromise() before a task completes', () => {
  const result = 'result-of-saga'

  const task = simpleRunSaga(function* saga() {
    yield delay(10)
    return result
  })
  expect(task.isRunning()).toBe(true)
  return expect(task.toPromise()).resolves.toBe(result)
})

test('calling toPromise() of an already aborted task', () => {
  const error = new Error('test-error')

  const task = simpleRunSaga(function* saga() {
    throw error
  })
  expect(task.isRunning()).toBe(false)
  return expect(task.toPromise()).rejects.toBe(error)
})

test('calling toPromise() before a task aborts', () => {
  const error = new Error('test-error')

  const task = simpleRunSaga(function* saga() {
    yield delay(10)
    throw error
  })
  expect(task.isRunning()).toBe(true)
  return expect(task.toPromise()).rejects.toBe(error)
})

test('calling toPromise() of an already cancelled task', async () => {
  let child

  simpleRunSaga(function* saga() {
    child = yield fork(function* child() {
      yield delay(10000)
    })
    yield cancel(child)
  })

  expect(child.isRunning()).toBe(false)
  return expect(child.toPromise()).resolves.toBe(TASK_CANCEL)
})

test('calling toPromise() of before a task gets cancelled', async () => {
  let child

  simpleRunSaga(function* saga() {
    child = yield fork(function* child() {
```

```
      yield delay(10000)
    })
    yield delay(10)
    yield cancel(child)
  })

  expect(child.isRunning()).toBe(true)
  return expect(child.toPromise()).resolves.toBe(TASK_CANCEL)
})
```

../redux-saga/packages/core/babel-transformer.jest.js

```
const path = require('path')
const { createTransformer } = require('babel-7-jest')

module.exports = createTransformer({
  babelrcRoots: path.resolve(__dirname, '../*'),
})
```

../redux-saga/packages/core/effects/effects.d.ts

```
export * from '../types/effects'
```

../redux-saga/packages/core/index.d.ts

```
export * from './types'
```

../redux-saga/packages/core/jest.config.js

```
const lernaAliases = require('lerna-alias').jest()

module.exports = {
  testEnvironment: 'node',
  moduleNameMapper: lernaAliases,
  transform: {
    '.js$': __dirname + '/babel-transformer.jest.js',
  },
}
```

../redux-saga/packages/core/rollup.config.js

```
import * as path from 'path'
import alias from 'rollup-plugin-alias'
import nodeResolve from 'rollup-plugin-node-resolve'
import babel from 'rollup-plugin-babel'
import replace from 'rollup-plugin-replace'
import { rollup as lernaAlias } from 'lerna-alias'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const helperPath = /^(@babel\/runtime\/helpers)\/(\w+)$/

const rewriteRuntimeHelpersImports = ({ types: t }) => ({
  name: 'rewrite-runtime-helpers-imports',
  visitor: {
    ImportDeclaration(path) {
      const source = path.get('source')
      if (!helperPath.test(source.node.value)) {
        return
      }
      const rewrittenPath = source.node.value.replace(helperPath, (m, p1, p2) => [p1, 'esm', p2].join('/'))
      source.replaceWith(t.stringLiteral(rewrittenPath))
    },
  },
})

const createConfig = ({ input, output, external, env, useESModules = output.format !== 'cjs' }) => ({
  input,
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(external === 'peers' ? peerDeps : deps.concat(peerDeps)),
  plugins: [
    alias(lernaAlias()),
    nodeResolve({
      jsnext: true,
    }),
    babel({
      exclude: 'node_modules/**',
      babelrcRoots: path.resolve(__dirname, '../*'),
      babelHelpers: 'runtime',
      plugins: [
        useESModules && rewriteRuntimeHelpersImports,
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
```

```
      ].filter(Boolean),
    }),
    env &&
      replace({
        'process.env.NODE_ENV': JSON.stringify(env),
      }),
  ].filter(Boolean),
  onwarn(warning, warn) {
    if (warning.code === 'UNUSED_EXTERNAL_IMPORT') {
      return
    }
    warn(warning)
  },
})

const multiInput = {
  core: 'src/index.js',
  effects: 'src/effects.js',
}

export default [
  createConfig({
    input: multiInput,
    output: {
      dir: 'dist',
      format: 'esm',
      entryFileNames: 'redux-saga-[name].[format].js',
    },
  }),
  createConfig({
    input: multiInput,
    output: {
      dir: 'dist',
      format: 'cjs',
      entryFileNames: 'redux-saga-[name].prod.[format].js',
    },
    env: 'production',
  }),
  createConfig({
    input: multiInput,
    output: {
      dir: 'dist',
      format: 'cjs',
      entryFileNames: 'redux-saga-[name].dev.[format].js',
    },
    env: 'development',
  }),
]
```

../redux-saga/packages/core/scripts/createProxyCjsEntries.js

```
const fs = require('fs')

const DIST_DIR = `${__dirname}/../dist`

const createEntryFile = (entry) => `"use strict";

if (process.env.NODE_ENV === 'production') {
  module.exports = require('./${entry}.prod.cjs.js')
} else {
  module.exports = require('./${entry}.dev.cjs.js')
}
`


const entries = fs
  .readdirSync(DIST_DIR)
  .filter((file) => /\.prod\.cjs\.js/.test(file))
  .map((file) => file.split('.')[0])

entries.forEach((entry) => fs.writeFileSync(`${DIST_DIR}/${entry}.cjs.js`, createEntryFile(entry), 'utf8'))
// eslint-disable-next-line no-console
console.log(`\tCreated proxy commonjs entries: ${entries.join(', ')}.\n`)
```

../redux-saga/packages/core/src/effects.js

```
export {
  take,
  takeMaybe,
  put,
  putResolve,
  all,
  race,
  call,
  apply,
  cps,
  fork,
  spawn,
  join,
  cancel,
  select,
  actionChannel,
  cancelled,
  flush,
  getContext,
  setContext,
  delay,
} from './internal/io'

export { debounce, retry, takeEvery, takeLatest, takeLeading, throttle } from './internal/io-helpers'

import * as effectTypes from './internal/effectTypes'

export { effectTypes }
```

../redux-saga/packages/core/src/index.js

```
export { CANCEL, SAGA_LOCATION } from '@redux-saga/symbols'
export { default } from './internal/middleware'

export { runSaga } from './internal/runSaga'
export { END, isEnd, eventChannel, channel, multicastChannel, stdChannel } from './internal/channel'

export { detach } from './internal/io'

import * as buffers from './internal/buffers'

export { buffers }
```

../redux-saga/packages/core/src/internal/buffers.js

```
import { kTrue, noop } from './utils'

const BUFFER_OVERFLOW = "Channel's Buffer overflow!"

const ON_OVERFLOW_THROW = 1
const ON_OVERFLOW_DROP = 2
const ON_OVERFLOW_SLIDE = 3
const ON_OVERFLOW_EXPAND = 4

const zeroBuffer = { isEmpty: kTrue, put: noop, take: noop }

function ringBuffer(limit = 10, overflowAction) {
  let arr = new Array(limit)
  let length = 0
  let pushIndex = 0
  let popIndex = 0

  const push = (it) => {
    arr[pushIndex] = it
    pushIndex = (pushIndex + 1) % limit
    length++
  }

  const take = () => {
    if (length != 0) {
      let it = arr[popIndex]
      arr[popIndex] = null
      length--
      popIndex = (popIndex + 1) % limit
      return it
    }
  }

  const flush = () => {
    let items = []
    while (length) {
      items.push(take())
    }
    return items
  }

  return {
    isEmpty: () => length == 0,
    put: (it) => {
      if (length < limit) {
        push(it)
      } else {
        let doubledLimit
        switch (overflowAction) {
          case ON_OVERFLOW_THROW:
            throw new Error(BUFFER_OVERFLOW)
          case ON_OVERFLOW_SLIDE:
            arr[pushIndex] = it
            pushIndex = (pushIndex + 1) % limit
            popIndex = pushIndex
            break
          case ON_OVERFLOW_EXPAND:
            doubledLimit = 2 * limit

            arr = flush()

            length = arr.length
            pushIndex = arr.length
            popIndex = 0

            arr.length = doubledLimit
            limit = doubledLimit

            push(it)
            break
          default:
          // DROP
        }
      }
    },
    take,
    flush,
  }
}

export const none = () => zeroBuffer
export const fixed = (limit) => ringBuffer(limit, ON_OVERFLOW_THROW)
export const dropping = (limit) => ringBuffer(limit, ON_OVERFLOW_DROP)
export const sliding = (limit) => ringBuffer(limit, ON_OVERFLOW_SLIDE)
export const expanding = (initialSize) => ringBuffer(initialSize, ON_OVERFLOW_EXPAND)
```

../redux-saga/packages/core/src/internal/channel.js

```
import * as is from '@redux-saga/is'
import { CHANNEL_END_TYPE, MATCH, MULTICAST, SAGA_ACTION } from '@redux-saga/symbols'
import { check, remove, once, internalErr } from './utils'
import * as buffers from './buffers'
import { asap } from './scheduler'
```

```
import * as matchers from './matcher'

export const END = { type: CHANNEL_END_TYPE }
export const isEnd = (a) => a && a.type === CHANNEL_END_TYPE

const CLOSED_CHANNEL_WITH_TAKERS = 'Cannot have a closed channel with pending takers'
const INVALID_BUFFER = 'invalid buffer passed to channel factory function'
const UNDEFINED_INPUT_ERROR = `Saga or channel was provided with an undefined action
Hints:
  - check that your Action Creator returns a non-undefined value
  - if the Saga was started using runSaga, check that your subscribe source provides the action to its listeners`

export function channel(buffer = buffers.expanding()) {
  let closed = false
  let takers = []

  if (process.env.NODE_ENV !== 'production') {
    check(buffer, is.buffer, INVALID_BUFFER)
  }

  function checkForbiddenStates() {
    if (closed && takers.length) {
      throw internalErr(CLOSED_CHANNEL_WITH_TAKERS)
    }
    if (takers.length && !buffer.isEmpty()) {
      throw internalErr('Cannot have pending takers with non empty buffer')
    }
  }

  function put(input) {
    if (process.env.NODE_ENV !== 'production') {
      checkForbiddenStates()
      check(input, is.notUndef, UNDEFINED_INPUT_ERROR)
    }

    if (closed) {
      return
    }
    if (takers.length === 0) {
      return buffer.put(input)
    }
    const cb = takers.shift()
    cb(input)
  }

  function take(cb) {
    if (process.env.NODE_ENV !== 'production') {
      checkForbiddenStates()
      check(cb, is.func, "channel.take's callback must be a function")
    }

    if (closed && buffer.isEmpty()) {
      cb(END)
    } else if (!buffer.isEmpty()) {
      cb(buffer.take())
    } else {
      takers.push(cb)
      cb.cancel = () => {
        remove(takers, cb)
      }
    }
  }

  function flush(cb) {
    if (process.env.NODE_ENV !== 'production') {
      checkForbiddenStates()
      check(cb, is.func, "channel.flush' callback must be a function")
    }

    if (closed && buffer.isEmpty()) {
      cb(END)
      return
    }
    cb(buffer.flush())
  }

  function close() {
    if (process.env.NODE_ENV !== 'production') {
      checkForbiddenStates()
    }

    if (closed) {
      return
    }

    closed = true

    const arr = takers
    takers = []

    for (let i = 0, len = arr.length; i < len; i++) {
      const taker = arr[i]
      taker(END)
    }
  }

  return {
    take,
    put,
    flush,
    close,
  }
}

export function eventChannel(subscribe, buffer = buffers.none()) {
  let closed = false
  let unsubscribe

  const chan = channel(buffer)
  const close = () => {
    if (closed) {
      return
```

```
    }

    closed = true

    if (is.func(unsubscribe)) {
      unsubscribe()
    }
    chan.close()
  }

  unsubscribe = subscribe((input) => {
    if (isEnd(input)) {
      close()
      return
    }
    chan.put(input)
  })

  if (process.env.NODE_ENV !== 'production') {
    check(unsubscribe, is.func, 'in eventChannel: subscribe should return a function to unsubscribe')
  }

  unsubscribe = once(unsubscribe)

  if (closed) {
    unsubscribe()
  }

  return {
    take: chan.take,
    flush: chan.flush,
    close,
  }
}

export function multicastChannel() {
  let closed = false
  let currentTakers = []
  let nextTakers = currentTakers

  function checkForbiddenStates() {
    if (closed && nextTakers.length) {
      throw internalErr(CLOSED_CHANNEL_WITH_TAKERS)
    }
  }

  const ensureCanMutateNextTakers = () => {
    if (nextTakers !== currentTakers) {
      return
    }
    nextTakers = currentTakers.slice()
  }

  const close = () => {
    if (process.env.NODE_ENV !== 'production') {
      checkForbiddenStates()
    }

    closed = true
    const takers = (currentTakers = nextTakers)
    nextTakers = []
    takers.forEach((taker) => {
      taker(END)
    })
  }

  return {
    [MULTICAST]: true,
    put(input) {
      if (process.env.NODE_ENV !== 'production') {
        checkForbiddenStates()
        check(input, is.notUndef, UNDEFINED_INPUT_ERROR)
      }

      if (closed) {
        return
      }

      if (isEnd(input)) {
        close()
        return
      }

      const takers = (currentTakers = nextTakers)

      for (let i = 0, len = takers.length; i < len; i++) {
        const taker = takers[i]

        if (taker[MATCH](input)) {
          taker.cancel()
          taker(input)
        }
      }
    },
    take(cb, matcher = matchers.wildcard) {
      if (process.env.NODE_ENV !== 'production') {
        checkForbiddenStates()
      }
      if (closed) {
        cb(END)
        return
      }
      cb[MATCH] = matcher
      ensureCanMutateNextTakers()
      nextTakers.push(cb)

      cb.cancel = once(() => {
        ensureCanMutateNextTakers()
        remove(nextTakers, cb)
      })
    },
    close,
```

```
  }
}

export function stdChannel() {
  const chan = multicastChannel()
  const { put } = chan
  chan.put = (input) => {
    if (input[SAGA_ACTION]) {
      put(input)
      return
    }
    asap(() => {
      put(input)
    })
  }
  return chan
}
```

../redux-saga/packages/core/src/internal/effectRunnerMap.js

```
import { SELF_CANCELLATION, TERMINATE } from '@redux-saga/symbols'
import * as is from '@redux-saga/is'
import * as effectTypes from './effectTypes'
import { channel, isEnd } from './channel'
// usage of proc here makes internal circular dependency
// this works fine, but it is a little bit unfortunate
import proc from './proc'
import resolvePromise from './resolvePromise'
import matcher from './matcher'
import { asap, immediately } from './scheduler'
import { current as currentEffectId } from './uid'
import {
  assignWithSymbols,
  createAllStyleChildCallbacks,
  createEmptyArray,
  makeIterator,
  noop,
  remove,
  shouldComplete,
  getMetaInfo,
} from './utils'

function getIteratorMetaInfo(iterator, fn) {
  if (iterator.isSagaIterator) {
    return { name: iterator.meta.name }
  }
  return getMetaInfo(fn)
}

function createTaskIterator({ context, fn, args }) {
  // catch synchronous failures; see #152 and #441
  try {
    const result = fn.apply(context, args)

    // i.e. a generator function returns an iterator
    if (is.iterator(result)) {
      return result
    }

    let resolved = false

    const next = (arg) => {
      if (!resolved) {
        resolved = true
        // Only promises returned from fork will be interpreted. See #1573
        return { value: result, done: !is.promise(result) }
      } else {
        return { value: arg, done: true }
      }
    }

    return makeIterator(next)
  } catch (err) {
    // do not bubble up synchronous failures for detached forks
    // instead create a failed task. See #152 and #441
    return makeIterator(() => {
      throw err
    })
  }
}

function runPutEffect(env, { channel, action, resolve }, cb) {
  /**
   Schedule the put in case another saga is holding a lock.
   The put will be executed atomically. ie nested puts will execute after
   this put has terminated.
   **/
  asap(() => {
    let result
    try {
      result = (channel ? channel.put : env.dispatch)(action)
    } catch (error) {
      cb(error, true)
      return
    }

    if (resolve && is.promise(result)) {
      resolvePromise(result, cb)
    } else {
      cb(result)
    }
  })
  // Put effects are non cancellables
}

function runTakeEffect(env, { channel = env.channel, pattern, maybe }, cb) {
  const takeCb = (input) => {
    if (input instanceof Error) {
      cb(input, true)
```

```
      return
    }
    if (isEnd(input) && !maybe) {
      cb(TERMINATE)
      return
    }
    cb(input)
  }
  try {
    channel.take(takeCb, is.notUndef(pattern) ? matcher(pattern) : null)
  } catch (err) {
    cb(err, true)
    return
  }
  cb.cancel = takeCb.cancel
}

function runCallEffect(env, { context, fn, args }, cb, { task }) {
  // catch synchronous failures; see #152
  try {
    const result = fn.apply(context, args)

    if (is.promise(result)) {
      resolvePromise(result, cb)
      return
    }

    if (is.iterator(result)) {
      // resolve iterator
      proc(env, result, task.context, currentEffectId, getMetaInfo(fn), /* isRoot */ false, cb)
      return
    }

    cb(result)
  } catch (error) {
    cb(error, true)
  }
}

function runCPSEffect(env, { context, fn, args }, cb) {
  // CPS (ie node style functions) can define their own cancellation logic
  // by setting cancel field on the cb

  // catch synchronous failures; see #152
  try {
    const cpsCb = (err, res) => {
      if (is.undef(err)) {
        cb(res)
      } else {
        cb(err, true)
      }
    }

    fn.apply(context, args.concat(cpsCb))

    if (cpsCb.cancel) {
      cb.cancel = cpsCb.cancel
    }
  } catch (error) {
    cb(error, true)
  }
}

function runForkEffect(env, { context, fn, args, detached }, cb, { task: parent }) {
  const taskIterator = createTaskIterator({ context, fn, args })
  const meta = getIteratorMetaInfo(taskIterator, fn)

  immediately(() => {
    const child = proc(env, taskIterator, parent.context, currentEffectId, meta, detached, undefined)

    if (detached) {
      cb(child)
    } else {
      if (child.isRunning()) {
        parent.queue.addTask(child)
        cb(child)
      } else if (child.isAborted()) {
        parent.queue.abort(child.error())
      } else {
        cb(child)
      }
    }
  })
  // Fork effects are non cancellables
}

function runJoinEffect(env, taskOrTasks, cb, { task }) {
  const joinSingleTask = (taskToJoin, cb) => {
    if (taskToJoin.isRunning()) {
      const joiner = { task, cb }
      cb.cancel = () => {
        if (taskToJoin.isRunning()) remove(taskToJoin.joiners, joiner)
      }
      taskToJoin.joiners.push(joiner)
    } else {
      if (taskToJoin.isAborted()) {
        cb(taskToJoin.error(), true)
      } else {
        cb(taskToJoin.result())
      }
    }
  }

  if (is.array(taskOrTasks)) {
    if (taskOrTasks.length === 0) {
      cb([])
      return
    }

    const childCallbacks = createAllStyleChildCallbacks(taskOrTasks, cb)
    taskOrTasks.forEach((t, i) => {
      joinSingleTask(t, childCallbacks[i])
```

```
    })
  } else {
    joinSingleTask(taskOrTasks, cb)
  }
}

function cancelSingleTask(taskToCancel) {
  if (taskToCancel.isRunning()) {
    taskToCancel.cancel()
  }
}

function runCancelEffect(env, taskOrTasks, cb, { task }) {
  if (taskOrTasks === SELF_CANCELLATION) {
    cancelSingleTask(task)
  } else if (is.array(taskOrTasks)) {
    taskOrTasks.forEach(cancelSingleTask)
  } else {
    cancelSingleTask(taskOrTasks)
  }
  cb()
  // cancel effects are non cancellables
}

function runAllEffect(env, effects, cb, { digestEffect }) {
  const effectId = currentEffectId
  const keys = Object.keys(effects)
  if (keys.length === 0) {
    cb(is.array(effects) ? [] : {})
    return
  }

  const childCallbacks = createAllStyleChildCallbacks(effects, cb)
  keys.forEach((key) => {
    digestEffect(effects[key], effectId, childCallbacks[key], key)
  })
}

function runRaceEffect(env, effects, cb, { digestEffect }) {
  const effectId = currentEffectId
  const keys = Object.keys(effects)
  const response = is.array(effects) ? createEmptyArray(keys.length) : {}
  const childCbs = {}
  let completed = false

  keys.forEach((key) => {
    const chCbAtKey = (res, isErr) => {
      if (completed) {
        return
      }
      if (isErr || shouldComplete(res)) {
        // Race Auto cancellation
        cb.cancel()
        cb(res, isErr)
      } else {
        cb.cancel()
        completed = true
        response[key] = res
        cb(response)
      }
    }
    chCbAtKey.cancel = noop
    childCbs[key] = chCbAtKey
  })

  cb.cancel = () => {
    // prevents unnecessary cancellation
    if (!completed) {
      completed = true
      keys.forEach((key) => childCbs[key].cancel())
    }
  }
  keys.forEach((key) => {
    if (completed) {
      return
    }
    digestEffect(effects[key], effectId, childCbs[key], key)
  })
}

function runSelectEffect(env, { selector, args }, cb) {
  try {
    const state = selector(env.getState(), ...args)
    cb(state)
  } catch (error) {
    cb(error, true)
  }
}

function runChannelEffect(env, { pattern, buffer }, cb) {
  const chan = channel(buffer)
  const match = matcher(pattern)

  const taker = (action) => {
    if (!isEnd(action)) {
      env.channel.take(taker, match)
    }
    chan.put(action)
  }

  const { close } = chan

  chan.close = () => {
    taker.cancel()
    close()
  }

  env.channel.take(taker, match)
  cb(chan)
}

function runCancelledEffect(env, data, cb, { task }) {
```

```
  cb(task.isCancelled())
}

function runFlushEffect(env, channel, cb) {
  channel.flush(cb)
}

function runGetContextEffect(env, prop, cb, { task }) {
  cb(task.context[prop])
}

function runSetContextEffect(env, props, cb, { task }) {
  assignWithSymbols(task.context, props)
  cb()
}

const effectRunnerMap = {
  [effectTypes.TAKE]: runTakeEffect,
  [effectTypes.PUT]: runPutEffect,
  [effectTypes.ALL]: runAllEffect,
  [effectTypes.RACE]: runRaceEffect,
  [effectTypes.CALL]: runCallEffect,
  [effectTypes.CPS]: runCPSEffect,
  [effectTypes.FORK]: runForkEffect,
  [effectTypes.JOIN]: runJoinEffect,
  [effectTypes.CANCEL]: runCancelEffect,
  [effectTypes.SELECT]: runSelectEffect,
  [effectTypes.ACTION_CHANNEL]: runChannelEffect,
  [effectTypes.CANCELLED]: runCancelledEffect,
  [effectTypes.FLUSH]: runFlushEffect,
  [effectTypes.GET_CONTEXT]: runGetContextEffect,
  [effectTypes.SET_CONTEXT]: runSetContextEffect,
}

export default effectRunnerMap
```

../redux-saga/packages/core/src/internal/effectTypes.js

```
export const TAKE = 'TAKE'
export const PUT = 'PUT'
export const ALL = 'ALL'
export const RACE = 'RACE'
export const CALL = 'CALL'
export const CPS = 'CPS'
export const FORK = 'FORK'
export const JOIN = 'JOIN'
export const CANCEL = 'CANCEL'
export const SELECT = 'SELECT'
export const ACTION_CHANNEL = 'ACTION_CHANNEL'
export const CANCELLED = 'CANCELLED'
export const FLUSH = 'FLUSH'
export const GET_CONTEXT = 'GET_CONTEXT'
export const SET_CONTEXT = 'SET_CONTEXT'
```

../redux-saga/packages/core/src/internal/forkQueue.js

```
import { noop, remove } from './utils'

/**
 Used to track a parent task and its forks
 In the fork model, forked tasks are attached by default to their parent
 We model this using the concept of Parent task && main Task
 main task is the main flow of the current Generator, the parent tasks is the
 aggregation of the main tasks + all its forked tasks.
 Thus the whole model represents an execution tree with multiple branches (vs the
 linear execution tree in sequential (non parallel) programming)

 A parent tasks has the following semantics
 - It completes if all its forks either complete or all cancelled
 - If it's cancelled, all forks are cancelled as well
 - It aborts if any uncaught error bubbles up from forks
 - If it completes, the return value is the one returned by the main task
 **/
export default function forkQueue(mainTask, onAbort, cont) {
  let tasks = []
  let result
  let completed = false

  addTask(mainTask)
  const getTasks = () => tasks

  function abort(err) {
    onAbort()
    cancelAll()
    cont(err, true)
  }

  function addTask(task) {
    tasks.push(task)
    task.cont = (res, isErr) => {
      if (completed) {
        return
      }

      remove(tasks, task)
      task.cont = noop
      if (isErr) {
        abort(res)
      } else {
        if (task === mainTask) {
          result = res
        }
        if (!tasks.length) {
          completed = true
          cont(result)
        }
      }
```

```
    }
  }

  function cancelAll() {
    if (completed) {
      return
    }
    completed = true
    tasks.forEach((t) => {
      t.cont = noop
      t.cancel()
    })
    tasks = []
  }

  return {
    addTask,
    cancelAll,
    abort,
    getTasks,
  }
}
```

../redux-saga/packages/core/src/internal/io-helpers.js

```
import * as is from '@redux-saga/is'
import { call, fork } from './io'
import { check } from './utils'
import {
  takeEveryHelper,
  takeLatestHelper,
  takeLeadingHelper,
  throttleHelper,
  retryHelper,
  debounceHelper,
} from './sagaHelpers'

const validateTakeEffect = (fn, patternOrChannel, worker) => {
  check(patternOrChannel, is.notUndef, `${fn.name} requires a pattern or channel`)
  check(worker, is.notUndef, `${fn.name} requires a saga parameter`)
}

export function takeEvery(patternOrChannel, worker, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateTakeEffect(takeEvery, patternOrChannel, worker)
  }

  return fork(takeEveryHelper, patternOrChannel, worker, ...args)
}

export function takeLatest(patternOrChannel, worker, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateTakeEffect(takeLatest, patternOrChannel, worker)
  }

  return fork(takeLatestHelper, patternOrChannel, worker, ...args)
}

export function takeLeading(patternOrChannel, worker, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateTakeEffect(takeLeading, patternOrChannel, worker)
  }

  return fork(takeLeadingHelper, patternOrChannel, worker, ...args)
}

export function throttle(ms, patternOrChannel, worker, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    check(patternOrChannel, is.notUndef, `throttle requires a pattern or channel`)
    check(worker, is.notUndef, 'throttle requires a saga parameter')
  }

  return fork(throttleHelper, ms, patternOrChannel, worker, ...args)
}

export function retry(maxTries, delayLength, worker, ...args) {
  return call(retryHelper, maxTries, delayLength, worker, ...args)
}

export function debounce(delayLength, pattern, worker, ...args) {
  return fork(debounceHelper, delayLength, pattern, worker, ...args)
}
```

../redux-saga/packages/core/src/internal/io.js

```
import delayP from '@redux-saga/delay-p'
import * as is from '@redux-saga/is'
import { IO, SELF_CANCELLATION } from '@redux-saga/symbols'
import { check, createSetContextWarning, identity } from './utils'
import * as effectTypes from './effectTypes'

const TEST_HINT =
  '\n(HINT: if you are getting these errors in tests, consider using createMockTask from @redux-saga/testing-utils)'

const makeEffect = (type, payload) => ({
  [IO]: true,
  // this property makes all/race distinguishable in generic manner from other effects
  // currently it's not used at runtime at all but it's here to satisfy type systems
  combinator: false,
  type,
  payload,
})

const isForkEffect = (eff) => is.effect(eff) && eff.type === effectTypes.FORK

export const detach = (eff) => {
  if (process.env.NODE_ENV !== 'production') {
```

```
    check(eff, isForkEffect, 'detach(eff): argument must be a fork effect')
  }
  return makeEffect(effectTypes.FORK, { ...eff.payload, detached: true })
}

export function take(patternOrChannel = '*', multicastPattern) {
  if (process.env.NODE_ENV !== 'production' && arguments.length) {
    check(arguments[0], is.notUndef, 'take(patternOrChannel): patternOrChannel is undefined')
  }
  if (is.pattern(patternOrChannel)) {
    if (is.notUndef(multicastPattern)) {
      /* eslint-disable no-console */
      console.warn(
        `take(pattern) takes one argument but two were provided. Consider passing an array for listening to several action types`,
      )
    }
    return makeEffect(effectTypes.TAKE, { pattern: patternOrChannel })
  }
  if (is.multicast(patternOrChannel) && is.notUndef(multicastPattern) && is.pattern(multicastPattern)) {
    return makeEffect(effectTypes.TAKE, { channel: patternOrChannel, pattern: multicastPattern })
  }
  if (is.channel(patternOrChannel)) {
    if (is.notUndef(multicastPattern)) {
      /* eslint-disable no-console */
      console.warn(`take(channel) takes one argument but two were provided. Second argument is ignored.`)
    }
    return makeEffect(effectTypes.TAKE, { channel: patternOrChannel })
  }
  if (process.env.NODE_ENV !== 'production') {
    throw new Error(`take(patternOrChannel): argument ${patternOrChannel} is not valid channel or a valid pattern`)
  }
}

export const takeMaybe = (...args) => {
  const eff = take(...args)
  eff.payload.maybe = true
  return eff
}

export function put(channel, action) {
  if (process.env.NODE_ENV !== 'production') {
    if (arguments.length > 1) {
      check(channel, is.notUndef, 'put(channel, action): argument channel is undefined')
      check(channel, is.channel, `put(channel, action): argument ${channel} is not a valid channel`)
      check(action, is.notUndef, 'put(channel, action): argument action is undefined')
    } else {
      check(channel, is.notUndef, 'put(action): argument action is undefined')
    }
  }
  if (is.undef(action)) {
    action = channel
    // `undefined` instead of `null` to make default parameter work
    channel = undefined
  }
  return makeEffect(effectTypes.PUT, { channel, action })
}

export const putResolve = (...args) => {
  const eff = put(...args)
  eff.payload.resolve = true
  return eff
}

export function all(effects) {
  const eff = makeEffect(effectTypes.ALL, effects)
  eff.combinator = true
  return eff
}

export function race(effects) {
  const eff = makeEffect(effectTypes.RACE, effects)
  eff.combinator = true
  return eff
}

// this match getFnCallDescriptor logic
const validateFnDescriptor = (effectName, fnDescriptor) => {
  check(fnDescriptor, is.notUndef, `${effectName}: argument fn is undefined or null`)

  if (is.func(fnDescriptor)) {
    return
  }

  let context = null
  let fn

  if (is.array(fnDescriptor)) {
    ;[context, fn] = fnDescriptor
    check(fn, is.notUndef, `${effectName}: argument of type [context, fn] has undefined or null \`fn\``)
  } else if (is.object(fnDescriptor)) {
    ;({ context, fn } = fnDescriptor)
    check(fn, is.notUndef, `${effectName}: argument of type {context, fn} has undefined or null \`fn\``)
  } else {
    check(fnDescriptor, is.func, `${effectName}: argument fn is not function`)
    return
  }

  if (context && is.string(fn)) {
    check(context[fn], is.func, `${effectName}: context arguments has no such method - "${fn}"`)
    return
  }

  check(fn, is.func, `${effectName}: unpacked fn argument (from [context, fn] or {context, fn}) is not a function`)
}

function getFnCallDescriptor(fnDescriptor, args) {
  let context = null
  let fn

  if (is.func(fnDescriptor)) {
    fn = fnDescriptor
  } else {
```

```
    if (is.array(fnDescriptor)) {
      ;[context, fn] = fnDescriptor
    } else {
      ;({ context, fn } = fnDescriptor)
    }

    if (context && is.string(fn) && is.func(context[fn])) {
      fn = context[fn]
    }
  }

  return { context, fn, args }
}

const isNotDelayEffect = (fn) => fn !== delay

export function call(fnDescriptor, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    const arg0 = typeof args[0] === 'number' ? args[0] : 'ms'
    check(
      fnDescriptor,
      isNotDelayEffect,
      `instead of writing \`yield call(delay, ${arg0})\` where delay is an effect from \`redux-saga/effects\` you should write \`yield delay(${arg0})\`
    )
    validateFnDescriptor('call', fnDescriptor)
  }
  return makeEffect(effectTypes.CALL, getFnCallDescriptor(fnDescriptor, args))
}

export function apply(context, fn, args = []) {
  const fnDescriptor = [context, fn]

  if (process.env.NODE_ENV !== 'production') {
    validateFnDescriptor('apply', fnDescriptor)
  }

  return makeEffect(effectTypes.CALL, getFnCallDescriptor([context, fn], args))
}

export function cps(fnDescriptor, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateFnDescriptor('cps', fnDescriptor)
  }
  return makeEffect(effectTypes.CPS, getFnCallDescriptor(fnDescriptor, args))
}

export function fork(fnDescriptor, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateFnDescriptor('fork', fnDescriptor)

    check(fnDescriptor, (arg) => !is.effect(arg), 'fork: argument must not be an effect')
  }
  return makeEffect(effectTypes.FORK, getFnCallDescriptor(fnDescriptor, args))
}

export function spawn(fnDescriptor, ...args) {
  if (process.env.NODE_ENV !== 'production') {
    validateFnDescriptor('spawn', fnDescriptor)
  }
  return detach(fork(fnDescriptor, ...args))
}

export function join(taskOrTasks) {
  if (process.env.NODE_ENV !== 'production') {
    if (arguments.length > 1) {
      throw new Error('join(...tasks) is not supported any more. Please use join([...tasks]) to join multiple tasks.')
    }
    if (is.array(taskOrTasks)) {
      taskOrTasks.forEach((t) => {
        check(t, is.task, `join([...tasks]): argument ${t} is not a valid Task object ${TEST_HINT}`)
      })
    } else {
      check(taskOrTasks, is.task, `join(task): argument ${taskOrTasks} is not a valid Task object ${TEST_HINT}`)
    }
  }

  return makeEffect(effectTypes.JOIN, taskOrTasks)
}

export function cancel(taskOrTasks = SELF_CANCELLATION) {
  if (process.env.NODE_ENV !== 'production') {
    if (arguments.length > 1) {
      throw new Error(
        'cancel(...tasks) is not supported any more. Please use cancel([...tasks]) to cancel multiple tasks.',
      )
    }
    if (is.array(taskOrTasks)) {
      taskOrTasks.forEach((t) => {
        check(t, is.task, `cancel([...tasks]): argument ${t} is not a valid Task object ${TEST_HINT}`)
      })
    } else if (taskOrTasks !== SELF_CANCELLATION && is.notUndef(taskOrTasks)) {
      check(taskOrTasks, is.task, `cancel(task): argument ${taskOrTasks} is not a valid Task object ${TEST_HINT}`)
    }
  }

  return makeEffect(effectTypes.CANCEL, taskOrTasks)
}

export function select(selector = identity, ...args) {
  if (process.env.NODE_ENV !== 'production' && arguments.length) {
    check(arguments[0], is.notUndef, 'select(selector, [...]): argument selector is undefined')
    check(selector, is.func, `select(selector, [...]): argument ${selector} is not a function`)
  }
  return makeEffect(effectTypes.SELECT, { selector, args })
}

/**
  channel(pattern, [buffer])    => creates a proxy channel for store actions
**/
export function actionChannel(pattern, buffer) {
  if (process.env.NODE_ENV !== 'production') {
    check(pattern, is.pattern, 'actionChannel(pattern,...): argument pattern is not valid')
```

```
    if (arguments.length > 1) {
      check(buffer, is.notUndef, 'actionChannel(pattern, buffer): argument buffer is undefined')
      check(buffer, is.buffer, `actionChannel(pattern, buffer): argument ${buffer} is not a valid buffer`)
    }
  }

  return makeEffect(effectTypes.ACTION_CHANNEL, { pattern, buffer })
}

export function cancelled() {
  return makeEffect(effectTypes.CANCELLED, {})
}

export function flush(channel) {
  if (process.env.NODE_ENV !== 'production') {
    check(channel, is.channel, `flush(channel): argument ${channel} is not valid channel`)
  }

  return makeEffect(effectTypes.FLUSH, channel)
}

export function getContext(prop) {
  if (process.env.NODE_ENV !== 'production') {
    check(prop, is.string, `getContext(prop): argument ${prop} is not a string`)
  }

  return makeEffect(effectTypes.GET_CONTEXT, prop)
}

export function setContext(props) {
  if (process.env.NODE_ENV !== 'production') {
    check(props, is.object, createSetContextWarning(null, props))
  }

  return makeEffect(effectTypes.SET_CONTEXT, props)
}

export const delay = call.bind(null, delayP)
```

../redux-saga/packages/core/src/internal/matcher.js

```
import * as is from '@redux-saga/is'
import { kTrue } from './utils'

export const array = (patterns) => (input) => patterns.some((p) => matcher(p)(input))
export const predicate = (predicate) => (input) => predicate(input)
export const string = (pattern) => (input) => input.type === String(pattern)
export const symbol = (pattern) => (input) => input.type === pattern
export const wildcard = () => kTrue

export default function matcher(pattern) {
  // prettier-ignore
  const matcherCreator = (
      pattern === '*'            ? wildcard
    : is.string(pattern)         ? string
    : is.array(pattern)          ? array
    : is.stringableFunc(pattern) ? string
    : is.func(pattern)           ? predicate
    : is.symbol(pattern)         ? symbol
    : null
  )

  if (matcherCreator === null) {
    throw new Error(`invalid pattern: ${pattern}`)
  }

  return matcherCreator(pattern)
}
```

../redux-saga/packages/core/src/internal/middleware.js

```
import * as is from '@redux-saga/is'
import { check, assignWithSymbols, createSetContextWarning } from './utils'
import { stdChannel } from './channel'
import { runSaga } from './runSaga'

export default function sagaMiddlewareFactory({ context = {}, channel = stdChannel(), sagaMonitor, ...options } = {}) {
  let boundRunSaga

  if (process.env.NODE_ENV !== 'production') {
    check(channel, is.channel, 'options.channel passed to the Saga middleware is not a channel')
  }

  function sagaMiddleware({ getState, dispatch }) {
    boundRunSaga = runSaga.bind(null, {
      ...options,
      context,
      channel,
      dispatch,
      getState,
      sagaMonitor,
    })

    return (next) => (action) => {
      if (sagaMonitor && sagaMonitor.actionDispatched) {
        sagaMonitor.actionDispatched(action)
      }
      const result = next(action) // hit reducers
      channel.put(action)
      return result
    }
  }

  sagaMiddleware.run = (...args) => {
    if (process.env.NODE_ENV !== 'production' && !boundRunSaga) {
      throw new Error('Before running a Saga, you must mount the Saga middleware on the Store using applyMiddleware')
```

```
    }
    return boundRunSaga(...args)
  }

  sagaMiddleware.setContext = (props) => {
    if (process.env.NODE_ENV !== 'production') {
      check(props, is.object, createSetContextWarning('sagaMiddleware', props))
    }

    assignWithSymbols(context, props)
  }

  return sagaMiddleware
}
```

../redux-saga/packages/core/src/internal/newTask.js

```
import deferred from '@redux-saga/deferred'
import * as is from '@redux-saga/is'
import { TASK, TASK_CANCEL } from '@redux-saga/symbols'
import { RUNNING, CANCELLED, ABORTED, DONE } from './task-status'
import { assignWithSymbols, check, createSetContextWarning, noop } from './utils'
import forkQueue from './forkQueue'
import * as sagaError from './sagaError'

export default function newTask(env, mainTask, parentContext, parentEffectId, meta, isRoot, cont = noop) {
  let status = RUNNING
  let taskResult
  let taskError
  let deferredEnd = null

  const cancelledDueToErrorTasks = []

  const context = Object.create(parentContext)
  const queue = forkQueue(
    mainTask,
    function onAbort() {
      cancelledDueToErrorTasks.push(...queue.getTasks().map((t) => t.meta.name))
    },
    end,
  )

  /**
   This may be called by a parent generator to trigger/propagate cancellation
   cancel all pending tasks (including the main task), then end the current task.

   Cancellation propagates down to the whole execution tree held by this Parent task
   It's also propagated to all joiners of this task and their execution tree/joiners

   Cancellation is noop for terminated/Cancelled tasks tasks
   **/
  function cancel() {
    if (status === RUNNING) {
      // Setting status to CANCELLED does not necessarily mean that the task/iterators are stopped
      // effects in the iterator's finally block will still be executed
      status = CANCELLED
      queue.cancelAll()
      // Ending with a TASK_CANCEL will propagate the Cancellation to all joiners
      end(TASK_CANCEL, false)
    }
  }

  function end(result, isErr) {
    if (!isErr) {
      // The status here may be RUNNING or CANCELLED
      // If the status is CANCELLED, then we do not need to change it here
      if (result === TASK_CANCEL) {
        status = CANCELLED
      } else if (status !== CANCELLED) {
        status = DONE
      }
      taskResult = result
      deferredEnd && deferredEnd.resolve(result)
    } else {
      status = ABORTED
      sagaError.addSagaFrame({ meta, cancelledTasks: cancelledDueToErrorTasks })

      if (task.isRoot) {
        const sagaStack = sagaError.toString()
        // we've dumped the saga stack to string and are passing it to user's code
        // we know that it won't be needed anymore and we need to clear it
        sagaError.clear()
        env.onError(result, { sagaStack })
      }
      taskError = result
      deferredEnd && deferredEnd.reject(result)
    }
    task.cont(result, isErr)
    task.joiners.forEach((joiner) => {
      joiner.cb(result, isErr)
    })
    task.joiners = null
  }

  function setContext(props) {
    if (process.env.NODE_ENV !== 'production') {
      check(props, is.object, createSetContextWarning('task', props))
    }

    assignWithSymbols(context, props)
  }

  function toPromise() {
    if (deferredEnd) {
      return deferredEnd.promise
    }

    deferredEnd = deferred()
```

```
    if (status === ABORTED) {
      deferredEnd.reject(taskError)
    } else if (status !== RUNNING) {
      deferredEnd.resolve(taskResult)
    }

    return deferredEnd.promise
  }

  const task = {
    // fields
    [TASK]: true,
    id: parentEffectId,
    meta,
    isRoot,
    context,
    joiners: [],
    queue,

    // methods
    cancel,
    cont,
    end,
    setContext,
    toPromise,
    isRunning: () => status === RUNNING,
    /*
      This method is used both for answering the cancellation status of the task and answering for CANCELLED effects.
      In most cases, the cancellation of a task propagates to all its unfinished children (including
      all forked tasks and the mainTask), so a naive implementation of this method would be:
        `() => status === CANCELLED || mainTask.status === CANCELLED`

      But there are cases that the task is aborted by an error and the abortion caused the mainTask to be cancelled.
      In such cases, the task is supposed to be aborted rather than cancelled, however the above naive implementation
      would return true for `task.isCancelled()`. So we need make sure that the task is running before accessing
      mainTask.status.

      There are cases that the task is cancelled when the mainTask is done (the task is waiting for forked children
      when cancellation occurs). In such cases, you may wonder `yield io.cancelled()` would return true because
      `status === CANCELLED` holds, and which is wrong. However, after the mainTask is done, the iterator cannot yield
      any further effects, so we can ignore such cases.

      See discussions in #1704
     */
    isCancelled: () => status === CANCELLED || (status === RUNNING && mainTask.status === CANCELLED),
    isAborted: () => status === ABORTED,
    result: () => taskResult,
    error: () => taskError,
  }

  return task
}
```

../redux-saga/packages/core/src/internal/proc.js

```
import * as is from '@redux-saga/is'
import { IO, TASK_CANCEL } from '@redux-saga/symbols'
import { RUNNING, CANCELLED, ABORTED, DONE } from './task-status'
import effectRunnerMap from './effectRunnerMap'
import resolvePromise from './resolvePromise'
import nextEffectId from './uid'
import { asyncIteratorSymbol, noop, shouldCancel, shouldTerminate } from './utils'
import newTask from './newTask'
import * as sagaError from './sagaError'

export default function proc(env, iterator, parentContext, parentEffectId, meta, isRoot, cont) {
  if (process.env.NODE_ENV !== 'production' && iterator[asyncIteratorSymbol]) {
    throw new Error("redux-saga doesn't support async generators, please use only regular ones")
  }

  const finalRunEffect = env.finalizeRunEffect(runEffect)

  /**
    Tracks the current effect cancellation
    Each time the generator progresses. calling runEffect will set a new value
    on it. It allows propagating cancellation to child effects
  **/
  next.cancel = noop

  /** Creates a main task to track the main flow */
  const mainTask = { meta, cancel: cancelMain, status: RUNNING }

  /**
   Creates a new task descriptor for this generator.
   A task is the aggregation of it's mainTask and all it's forked tasks.
   **/
  const task = newTask(env, mainTask, parentContext, parentEffectId, meta, isRoot, cont)

  const executingContext = {
    task,
    digestEffect,
  }

  /**
    cancellation of the main task. We'll simply resume the Generator with a TASK_CANCEL
  **/
  function cancelMain() {
    if (mainTask.status === RUNNING) {
      mainTask.status = CANCELLED
      next(TASK_CANCEL)
    }
  }

  /**
    attaches cancellation logic to this task's continuation
    this will permit cancellation to propagate down the call chain
  **/
  if (cont) {
    cont.cancel = task.cancel
```

```
}

// kicks up the generator
next()

// then return the task descriptor to the caller
return task

/**
 * This is the generator driver
 * It's a recursive async/continuation function which calls itself
 * until the generator terminates or throws
 * @param {internal commands(TASK_CANCEL | TERMINATE) | any} arg - value, generator will be resumed with.
 * @param {boolean} isErr - the flag shows if effect finished with an error
 *
 * receives either (command | effect result, false) or (any thrown thing, true)
 */
function next(arg, isErr) {
  try {
    let result
    if (isErr) {
      result = iterator.throw(arg)
      // user handled the error, we can clear bookkept values
      sagaError.clear()
    } else if (shouldCancel(arg)) {
      /**
        getting TASK_CANCEL automatically cancels the main task
        We can get this value here

        - By cancelling the parent task manually
        - By joining a Cancelled task
      **/
      mainTask.status = CANCELLED
      /**
        Cancels the current effect; this will propagate the cancellation down to any called tasks
      **/
      next.cancel()
      /**
        If this Generator has a `return` method then invokes it
        This will jump to the finally block
      **/
      result = is.func(iterator.return) ? iterator.return(TASK_CANCEL) : { done: true, value: TASK_CANCEL }
    } else if (shouldTerminate(arg)) {
      // We get TERMINATE flag, i.e. by taking from a channel that ended using `take` (and not `takem` used to trap End of channels)
      result = is.func(iterator.return) ? iterator.return() : { done: true }
    } else {
      result = iterator.next(arg)
    }

    if (!result.done) {
      digestEffect(result.value, parentEffectId, next)
    } else {
      /**
        This Generator has ended, terminate the main task and notify the fork queue
      **/
      if (mainTask.status !== CANCELLED) {
        mainTask.status = DONE
      }
      mainTask.cont(result.value)
    }
  } catch (error) {
    if (mainTask.status === CANCELLED) {
      throw error
    }
    mainTask.status = ABORTED

    mainTask.cont(error, true)
  }
}

function runEffect(effect, effectId, currCb) {
  /**
    each effect runner must attach its own logic of cancellation to the provided callback
    it allows this generator to propagate cancellation downward.

    ATTENTION! effect runners must setup the cancel logic by setting cb.cancel = [cancelMethod]
    And the setup must occur before calling the callback

    This is a sort of inversion of control: called async functions are responsible
    of completing the flow by calling the provided continuation; while caller functions
    are responsible for aborting the current flow by calling the attached cancel function

    Library users can attach their own cancellation logic to promises by defining a
    promise[CANCEL] method in their returned promises
    ATTENTION! calling cancel must have no effect on an already completed or cancelled effect
  **/
  if (is.promise(effect)) {
    resolvePromise(effect, currCb)
  } else if (is.iterator(effect)) {
    // resolve iterator
    proc(env, effect, task.context, effectId, meta, /* isRoot */ false, currCb)
  } else if (effect && effect[IO]) {
    const effectRunner = effectRunnerMap[effect.type]
    effectRunner(env, effect.payload, currCb, executingContext)
  } else {
    // anything else returned as is
    currCb(effect)
  }
}

function digestEffect(effect, parentEffectId, cb, label = '') {
  const effectId = nextEffectId()
  env.sagaMonitor && env.sagaMonitor.effectTriggered({ effectId, parentEffectId, label, effect })

  /**
    completion callback and cancel callback are mutually exclusive
    We can't cancel an already completed effect
    And We can't complete an already cancelled effectId
  **/
  let effectSettled

  // Completion callback passed to the appropriate effect runner
```

```
    function currCb(res, isErr) {
      if (effectSettled) {
        return
      }

      effectSettled = true
      cb.cancel = noop // defensive measure
      if (env.sagaMonitor) {
        if (isErr) {
          env.sagaMonitor.effectRejected(effectId, res)
        } else {
          env.sagaMonitor.effectResolved(effectId, res)
        }
      }

      if (isErr) {
        sagaError.setCrashedEffect(effect)
      }

      cb(res, isErr)
    }
    // tracks down the current cancel
    currCb.cancel = noop

    // setup cancellation logic on the parent cb
    cb.cancel = () => {
      // prevents cancelling an already completed effect
      if (effectSettled) {
        return
      }

      effectSettled = true

      currCb.cancel() // propagates cancel downward
      currCb.cancel = noop // defensive measure

      env.sagaMonitor && env.sagaMonitor.effectCancelled(effectId)
    }

    finalRunEffect(effect, effectId, currCb)
  }
}
```

../redux-saga/packages/core/src/internal/resolvePromise.js

```
import * as is from '@redux-saga/is'
import { CANCEL } from '@redux-saga/symbols'

export default function resolvePromise(promise, cb) {
  const cancelPromise = promise[CANCEL]

  if (is.func(cancelPromise)) {
    cb.cancel = cancelPromise
  }

  promise.then(cb, (error) => {
    cb(error, true)
  })
}
```

../redux-saga/packages/core/src/internal/runSaga.js

```
import * as is from '@redux-saga/is'
import proc from './proc'
import { stdChannel } from './channel'
import { immediately } from './scheduler'
import nextSagaId from './uid'
import { check, logError, noop, wrapSagaDispatch, identity, getMetaInfo, compose } from './utils'

const RUN_SAGA_SIGNATURE = 'runSaga(options, saga, ...args)'
const NON_GENERATOR_ERR = `${RUN_SAGA_SIGNATURE}: saga argument must be a Generator function!`

export function runSaga(
  { channel = stdChannel(), dispatch, getState, context = {}, sagaMonitor, effectMiddlewares, onError = logError },
  saga,
  ...args
) {
  if (process.env.NODE_ENV !== 'production') {
    check(saga, is.func, NON_GENERATOR_ERR)
  }

  const iterator = saga(...args)

  if (process.env.NODE_ENV !== 'production') {
    check(iterator, is.iterator, NON_GENERATOR_ERR)
  }

  const effectId = nextSagaId()

  if (sagaMonitor) {
    // monitors are expected to have a certain interface, let's fill-in any missing ones
    sagaMonitor.rootSagaStarted = sagaMonitor.rootSagaStarted || noop
    sagaMonitor.effectTriggered = sagaMonitor.effectTriggered || noop
    sagaMonitor.effectResolved = sagaMonitor.effectResolved || noop
    sagaMonitor.effectRejected = sagaMonitor.effectRejected || noop
    sagaMonitor.effectCancelled = sagaMonitor.effectCancelled || noop
    sagaMonitor.actionDispatched = sagaMonitor.actionDispatched || noop

    sagaMonitor.rootSagaStarted({ effectId, saga, args })
  }

  if (process.env.NODE_ENV !== 'production') {
    if (is.notUndef(dispatch)) {
      check(dispatch, is.func, 'dispatch must be a function')
    }

    if (is.notUndef(getState)) {
```

```
    check(getState, is.func, 'getState must be a function')
  }

  if (is.notUndef(effectMiddlewares)) {
    const MIDDLEWARE_TYPE_ERROR = 'effectMiddlewares must be an array of functions'
    check(effectMiddlewares, is.array, MIDDLEWARE_TYPE_ERROR)
    effectMiddlewares.forEach((effectMiddleware) => check(effectMiddleware, is.func, MIDDLEWARE_TYPE_ERROR))
  }

  check(onError, is.func, 'onError passed to the redux-saga is not a function!')
}

let finalizeRunEffect
if (effectMiddlewares) {
  const middleware = compose(...effectMiddlewares)
  finalizeRunEffect = (runEffect) => {
    return (effect, effectId, currCb) => {
      const plainRunEffect = (eff) => runEffect(eff, effectId, currCb)
      return middleware(plainRunEffect)(effect)
    }
  }
} else {
  finalizeRunEffect = identity
}

const env = {
  channel,
  dispatch: wrapSagaDispatch(dispatch),
  getState,
  sagaMonitor,
  onError,
  finalizeRunEffect,
}

return immediately(() => {
  const task = proc(env, iterator, context, effectId, getMetaInfo(saga), /* isRoot */ true, undefined)

  if (sagaMonitor) {
    sagaMonitor.effectResolved(effectId, task)
  }

  return task
})
}
```

../redux-saga/packages/core/src/internal/sagaError.js

```
// there can be only a single saga error created at any given moment
// so this module acts as a singleton for bookkeeping it
import { getLocation, flatMap } from './utils'

function formatLocation(fileName, lineNumber) {
  return `${fileName}?${lineNumber}`
}

function effectLocationAsString(effect) {
  const location = getLocation(effect)
  if (location) {
    const { code, fileName, lineNumber } = location
    const source = `${code}  ${formatLocation(fileName, lineNumber)}`
    return source
  }
  return ''
}

function sagaLocationAsString(sagaMeta) {
  const { name, location } = sagaMeta
  if (location) {
    return `${name}  ${formatLocation(location.fileName, location.lineNumber)}`
  }
  return name
}

function cancelledTasksAsString(sagaStack) {
  const cancelledTasks = flatMap((i) => i.cancelledTasks, sagaStack)
  if (!cancelledTasks.length) {
    return ''
  }
  return ['Tasks cancelled due to error:', ...cancelledTasks].join('\n')
}

let crashedEffect = null
const sagaStack = []

export const addSagaFrame = (frame) => {
  frame.crashedEffect = crashedEffect
  sagaStack.push(frame)
}

export const clear = () => {
  crashedEffect = null
  sagaStack.length = 0
}

// this sets crashed effect for the soon-to-be-reported saga frame
// this slightly streatches the singleton nature of this module into wrong direction
// as it's even less obvious what's the data flow here, but it is what it is for now
export const setCrashedEffect = (effect) => {
  crashedEffect = effect
}

/**
  @returns {string}

  @example
  The above error occurred in task errorInPutSaga {pathToFile}
  when executing effect put({type: 'REDUCER_ACTION_ERROR_IN_PUT'}) {pathToFile}
      created by fetchSaga {pathToFile}
      created by rootSaga {pathToFile}
```

```
*/
export const toString = () => {
  const [firstSaga, ...otherSagas] = sagaStack
  const crashedEffectLocation = firstSaga.crashedEffect ? effectLocationAsString(firstSaga.crashedEffect) : null
  const errorMessage = `The above error occurred in task ${sagaLocationAsString(firstSaga.meta)}${
    crashedEffectLocation ? ` \n when executing effect ${crashedEffectLocation}` : ''
  }`

  return [
    errorMessage,
    ...otherSagas.map((s) => `    created by ${sagaLocationAsString(s.meta)}`),
    cancelledTasksAsString(sagaStack),
  ].join('\n')
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/debounce.js

```
import fsmIterator, { safeName } from './fsmIterator'
import { delay, fork, race, take } from '../io'

export default function debounceHelper(delayLength, patternOrChannel, worker, ...args) {
  let action, raceOutput

  const yTake = { done: false, value: take(patternOrChannel) }
  const yRace = {
    done: false,
    value: race({
      action: take(patternOrChannel),
      debounce: delay(delayLength),
    }),
  }
  const yFork = (ac) => ({ done: false, value: fork(worker, ...args, ac) })
  const yNoop = (value) => ({ done: false, value })

  const setAction = (ac) => (action = ac)
  const setRaceOutput = (ro) => (raceOutput = ro)

  return fsmIterator(
    {
      q1() {
        return { nextState: 'q2', effect: yTake, stateUpdater: setAction }
      },
      q2() {
        return { nextState: 'q3', effect: yRace, stateUpdater: setRaceOutput }
      },
      q3() {
        return raceOutput.debounce
          ? { nextState: 'q1', effect: yFork(action) }
          : { nextState: 'q2', effect: yNoop(raceOutput.action), stateUpdater: setAction }
      },
    },
    'q1',
    `debounce(${safeName(patternOrChannel)}, ${worker.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/fsmIterator.js

```
import * as is from '@redux-saga/is'
import { makeIterator } from '../utils'

const done = (value) => ({ done: true, value })
export const qEnd = {}

export function safeName(patternOrChannel) {
  if (is.channel(patternOrChannel)) {
    return 'channel'
  }

  if (is.stringableFunc(patternOrChannel)) {
    return String(patternOrChannel)
  }

  if (is.func(patternOrChannel)) {
    return patternOrChannel.name
  }

  return String(patternOrChannel)
}

export default function fsmIterator(fsm, startState, name) {
  let stateUpdater,
    errorState,
    effect,
    nextState = startState

  function next(arg, error) {
    if (nextState === qEnd) {
      return done(arg)
    }
    if (error && !errorState) {
      nextState = qEnd
      throw error
    } else {
      stateUpdater && stateUpdater(arg)
      const currentState = error ? fsm[errorState](error) : fsm[nextState]()
      ;({ nextState, effect, stateUpdater, errorState } = currentState)
      return nextState === qEnd ? done(arg) : effect
    }
  }

  return makeIterator(next, (error) => next(null, error), name)
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/index.js

```
export { default as takeEveryHelper } from './takeEvery'
export { default as takeLatestHelper } from './takeLatest'
export { default as takeLeadingHelper } from './takeLeading'
export { default as throttleHelper } from './throttle'
export { default as retryHelper } from './retry'
export { default as debounceHelper } from './debounce'
```

../redux-saga/packages/core/src/internal/sagaHelpers/retry.js

```
import fsmIterator, { qEnd } from './fsmIterator'
import { call, delay } from '../io'

export default function retry(maxTries, delayLength, fn, ...args) {
  let counter = maxTries

  const yCall = { done: false, value: call(fn, ...args) }
  const yDelay = { done: false, value: delay(delayLength) }

  return fsmIterator(
    {
      q1() {
        return { nextState: 'q2', effect: yCall, errorState: 'q10' }
      },
      q2() {
        return { nextState: qEnd }
      },
      q10(error) {
        counter -= 1
        if (counter <= 0) {
          throw error
        }
        return { nextState: 'q1', effect: yDelay }
      },
    },
    'q1',
    `retry(${fn.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/takeEvery.js

```
import fsmIterator, { safeName } from './fsmIterator'
import { take, fork } from '../io'

export default function takeEvery(patternOrChannel, worker, ...args) {
  const yTake = { done: false, value: take(patternOrChannel) }
  const yFork = (ac) => ({ done: false, value: fork(worker, ...args, ac) })

  let action,
    setAction = (ac) => (action = ac)

  return fsmIterator(
    {
      q1() {
        return { nextState: 'q2', effect: yTake, stateUpdater: setAction }
      },
      q2() {
        return { nextState: 'q1', effect: yFork(action) }
      },
    },
    'q1',
    `takeEvery(${safeName(patternOrChannel)}, ${worker.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/takeLatest.js

```
import fsmIterator, { safeName } from './fsmIterator'
import { cancel, take, fork } from '../io'

export default function takeLatest(patternOrChannel, worker, ...args) {
  const yTake = { done: false, value: take(patternOrChannel) }
  const yFork = (ac) => ({ done: false, value: fork(worker, ...args, ac) })
  const yCancel = (task) => ({ done: false, value: cancel(task) })

  let task, action
  const setTask = (t) => (task = t)
  const setAction = (ac) => (action = ac)

  return fsmIterator(
    {
      q1() {
        return { nextState: 'q2', effect: yTake, stateUpdater: setAction }
      },
      q2() {
        return task
          ? { nextState: 'q3', effect: yCancel(task) }
          : { nextState: 'q1', effect: yFork(action), stateUpdater: setTask }
      },
      q3() {
        return { nextState: 'q1', effect: yFork(action), stateUpdater: setTask }
      },
    },
    'q1',
    `takeLatest(${safeName(patternOrChannel)}, ${worker.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/takeLeading.js

```
import fsmIterator, { safeName } from './fsmIterator'
import { take, call } from '../io'
```

```
export default function takeLeading(patternOrChannel, worker, ...args) {
  const yTake = { done: false, value: take(patternOrChannel) }
  const yCall = (ac) => ({ done: false, value: call(worker, ...args, ac) })

  let action
  const setAction = (ac) => (action = ac)

  return fsmIterator(
    {
      q1() {
        return { nextState: 'q2', effect: yTake, stateUpdater: setAction }
      },
      q2() {
        return { nextState: 'q1', effect: yCall(action) }
      },
    },
    'q1',
    `takeLeading(${safeName(patternOrChannel)}, ${worker.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/sagaHelpers/throttle.js

```
import * as is from '@redux-saga/is'
import fsmIterator, { safeName } from './fsmIterator'
import { take, fork, actionChannel, delay } from '../io'
import * as buffers from '../buffers'

export default function throttle(delayLength, patternOrChannel, worker, ...args) {
  let action, channel

  const yTake = () => ({ done: false, value: take(channel) })
  const yFork = (ac) => ({ done: false, value: fork(worker, ...args, ac) })
  const yDelay = { done: false, value: delay(delayLength) }

  const setAction = (ac) => (action = ac)
  const setChannel = (ch) => (channel = ch)

  const needsChannel = !is.channel(patternOrChannel)

  if (!needsChannel) {
    setChannel(patternOrChannel)
  }

  return fsmIterator(
    {
      q1() {
        const yActionChannel = { done: false, value: actionChannel(patternOrChannel, buffers.sliding(1)) }
        return { nextState: 'q2', effect: yActionChannel, stateUpdater: setChannel }
      },
      q2() {
        return { nextState: 'q3', effect: yTake(), stateUpdater: setAction }
      },
      q3() {
        return { nextState: 'q4', effect: yFork(action) }
      },
      q4() {
        return { nextState: 'q2', effect: yDelay }
      },
    },
    needsChannel ? 'q1' : 'q2',
    `throttle(${safeName(patternOrChannel)}, ${worker.name})`,
  )
}
```

../redux-saga/packages/core/src/internal/scheduler.js

```
const queue = []
/**
  Variable to hold a counting semaphore
  - Incrementing adds a lock and puts the scheduler in a `suspended` state (if it's not
    already suspended)
  - Decrementing releases a lock. Zero locks puts the scheduler in a `released` state. This
    triggers flushing the queued tasks.
**/
let semaphore = 0

/**
  Executes a task 'atomically'. Tasks scheduled during this execution will be queued
  and flushed after this task has finished (assuming the scheduler endup in a released
  state).
**/
function exec(task) {
  try {
    suspend()
    task()
  } finally {
    release()
  }
}

/**
  Executes or queues a task depending on the state of the scheduler (`suspended` or `released`)
**/
export function asap(task) {
  queue.push(task)

  if (!semaphore) {
    suspend()
    flush()
  }
}

/**
 * Puts the scheduler in a `suspended` state and executes a task immediately.
 */
```

```
export function immediately(task) {
  try {
    suspend()
    return task()
  } finally {
    flush()
  }
}

/**
  Puts the scheduler in a `suspended` state. Scheduled tasks will be queued until the
  scheduler is released.
**/
function suspend() {
  semaphore++
}

/**
  Puts the scheduler in a `released` state.
**/
function release() {
  semaphore--
}

/**
  Releases the current lock. Executes all queued tasks if the scheduler is in the released state.
**/
function flush() {
  release()

  let task
  while (!semaphore && (task = queue.shift()) !== undefined) {
    exec(task)
  }
}
```

../redux-saga/packages/core/src/internal/task-status.js

```
export const RUNNING = 0
export const CANCELLED = 1
export const ABORTED = 2
export const DONE = 3
```

../redux-saga/packages/core/src/internal/uid.js

```
export let current = 0

export default () => ++current
```

../redux-saga/packages/core/src/internal/utils.js

```
import _extends from '@babel/runtime/helpers/extends'
import * as is from '@redux-saga/is'
import { SAGA_LOCATION, SAGA_ACTION, TASK_CANCEL, TERMINATE } from '@redux-saga/symbols'

export const konst = (v) => () => v
export const kTrue = konst(true)
export const kFalse = konst(false)

let noop = () => {}

if (process.env.NODE_ENV !== 'production' && typeof Proxy !== 'undefined') {
  noop = new Proxy(noop, {
    set: () => {
      throw internalErr('There was an attempt to assign a property to internal `noop` function.')
    },
  })
}

export { noop }

export const identity = (v) => v

const hasSymbol = typeof Symbol === 'function'
export const asyncIteratorSymbol = hasSymbol && Symbol.asyncIterator ? Symbol.asyncIterator : '@@asyncIterator'

export function check(value, predicate, error) {
  if (!predicate(value)) {
    throw new Error(error)
  }
}

const hasOwnProperty = Object.prototype.hasOwnProperty
export function hasOwn(object, property) {
  return is.notUndef(object) && hasOwnProperty.call(object, property)
}

export const assignWithSymbols = (target, source) => {
  _extends(target, source)

  if (Object.getOwnPropertySymbols) {
    Object.getOwnPropertySymbols(source).forEach((s) => {
      target[s] = source[s]
    })
  }
}

export const flatMap = (mapper, arr) => [].concat(...arr.map(mapper))

export function remove(array, item) {
  const index = array.indexOf(item)
  if (index >= 0) {
    array.splice(index, 1)
  }
}
```

```
export function once(fn) {
  let called = false
  return () => {
    if (called) {
      return
    }
    called = true
    fn()
  }
}

const kThrow = (err) => {
  throw err
}
const kReturn = (value) => ({ value, done: true })
export function makeIterator(next, thro = kThrow, name = 'iterator') {
  const iterator = { meta: { name }, next, throw: thro, return: kReturn, isSagaIterator: true }

  if (typeof Symbol !== 'undefined') {
    iterator[Symbol.iterator] = () => iterator
  }
  return iterator
}

export function logError(error, { sagaStack }) {
  /*eslint-disable no-console*/
  console.error(error)
  console.error(sagaStack)
}

export function deprecate(fn, deprecationWarning) {
  return (...args) => {
    if (process.env.NODE_ENV !== 'production') console.warn(deprecationWarning)
    return fn(...args)
  }
}

export const internalErr = (err) =>
  new Error(
    `
  redux-saga: Error checking hooks detected an inconsistent state. This is likely a bug
  in redux-saga code and not yours. Thanks for reporting this in the project's github repo.
  Error: ${err}
`,
  )

export const createSetContextWarning = (ctx, props) =>
  `${ctx ? ctx + '.' : ''}setContext(props): argument ${props} is not a plain object`

const FROZEN_ACTION_ERROR = `You can't put (a.k.a. dispatch from saga) frozen actions.
We have to define a special non-enumerable property on those actions for scheduling purposes.
Otherwise you wouldn't be able to communicate properly between sagas & other subscribers (action ordering would become far less predictable).
If you are using redux and you care about this behaviour (frozen actions),
then you might want to switch to freezing actions in a middleware rather than in action creator.
Example implementation:

const freezeActions = store => next => action => next(Object.freeze(action))
`

// creates empty, but not-holey array
export const createEmptyArray = (n) => Array.apply(null, new Array(n))

export const wrapSagaDispatch = (dispatch) => (action) => {
  if (process.env.NODE_ENV !== 'production') {
    check(action, (ac) => !Object.isFrozen(ac), FROZEN_ACTION_ERROR)
  }
  return dispatch(Object.defineProperty(action, SAGA_ACTION, { value: true }))
}

export const shouldTerminate = (res) => res === TERMINATE
export const shouldCancel = (res) => res === TASK_CANCEL
export const shouldComplete = (res) => shouldTerminate(res) || shouldCancel(res)

export function createAllStyleChildCallbacks(shape, parentCallback) {
  const keys = Object.keys(shape)
  const totalCount = keys.length

  if (process.env.NODE_ENV !== 'production') {
    check(totalCount, (c) => c > 0, 'createAllStyleChildCallbacks: get an empty array or object')
  }

  let completedCount = 0
  let completed
  const results = is.array(shape) ? createEmptyArray(totalCount) : {}
  const childCallbacks = {}

  function checkEnd() {
    if (completedCount === totalCount) {
      completed = true
      parentCallback(results)
    }
  }

  keys.forEach((key) => {
    const chCbAtKey = (res, isErr) => {
      if (completed) {
        return
      }
      if (isErr || shouldComplete(res)) {
        parentCallback.cancel()
        parentCallback(res, isErr)
      } else {
        results[key] = res
        completedCount++
        checkEnd()
      }
    }
    chCbAtKey.cancel = noop
    childCallbacks[key] = chCbAtKey
  })
```

```
  parentCallback.cancel = () => {
    if (!completed) {
      completed = true
      keys.forEach((key) => childCallbacks[key].cancel())
    }
  }

  return childCallbacks
}

export function getMetaInfo(fn) {
  return {
    name: fn.name || 'anonymous',
    location: getLocation(fn),
  }
}

export function getLocation(instrumented) {
  return instrumented[SAGA_LOCATION]
}

export function compose(...funcs) {
  if (funcs.length === 0) {
    return (arg) => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce(
    (a, b) =>
      (...args) =>
        a(b(...args)),
  )
}
```

../redux-saga/packages/core/types/channels.test.ts

```
import {
  buffers, Buffer, channel, Channel, EventChannel, MulticastChannel, END,
  eventChannel, multicastChannel, stdChannel,
} from "redux-saga";

function testBuffers() {
  const b1: Buffer<{foo: string}> = buffers.none<{foo: string}>();

  const b2: Buffer<{foo: string}> = buffers.dropping<{foo: string}>();
  const b3: Buffer<{foo: string}> = buffers.dropping<{foo: string}>(42);

  const b4: Buffer<{foo: string}> = buffers.expanding<{foo: string}>();
  const b5: Buffer<{foo: string}> = buffers.expanding<{foo: string}>(42);

  const b6: Buffer<{foo: string}> = buffers.fixed<{foo: string}>();
  const b7: Buffer<{foo: string}> = buffers.fixed<{foo: string}>(42);

  const b8: Buffer<{foo: string}> = buffers.sliding<{foo: string}>();
  const b9: Buffer<{foo: string}> = buffers.sliding<{foo: string}>(42);

  const buffer = buffers.none<{foo: string}>();

  // $ExpectError
  buffer.put({bar: 'bar'});
  buffer.put({foo: 'foo'});

  const isEmpty: boolean = buffer.isEmpty();

  const item = buffer.take();

  // $ExpectError
  item.foo;  // item may be undefined

  const foo: string = item!.foo;

  if (buffer.flush)
    buffer.flush();
}

function testChannel() {
  const c1: Channel<{foo: string}> = channel<{foo: string}>();
  const c2: Channel<{foo: string}> = channel(buffers.none<{foo: string}>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);

  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: Array<{bar: number}> | END) => {});
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();

  // Testing that we can't define channels that pass void or undefined
  // $ExpectError
  const voidChannel: Channel<void> = channel();
  // $ExpectError
  const voidChannel2 = channel<void>();
  // $ExpectError
  const undefinedChannel = channel<undefined>();
  // $ExpectError
```

```
  channel().put();
  // $ExpectError
  channel().put(undefined);

  // Testing that we can pass primitives into channels
  channel().put(42);
  channel().put('test');
  channel().put(true);
}

function testEventChannel(secs: number) {
  const subscribe = (emitter: (input: number | END) => void) => {
    const iv = setInterval(() => {
      secs -= 1
      if (secs > 0) {
        emitter(secs)
      } else {
        emitter(END)
        clearInterval(iv)
      }
    }, 1000);
    return () => {
      clearInterval(iv)
    }
  };
  const c1: EventChannel<number> = eventChannel<number>(subscribe);

  const c2: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<string>()); // $ExpectError

  const c3: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<number>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: string | END) => {});
  c1.take((message: number | END) => {});

  // $ExpectError
  c1.put(1);

  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: string[] | END) => {});
  c1.flush((messages: number[] | END) => {});

  c1.close();

  // $ExpectError
  const c4: EventChannel<void> = eventChannel(() => () => {})

  // $ExpectError
  const c5 = eventChannel<void>(emit => {
    emit()
    return () => {}
  })

  const c6 = eventChannel(emit => {
    // $ExpectError
    emit()
    return () => {}
  })
}

function testMulticastChannel() {
  const c1: MulticastChannel<{foo: string}> = multicastChannel<{foo: string}>();
  const c2: MulticastChannel<{foo: string}> = stdChannel<{foo: string}>();

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);

  // $ExpectError
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();

  // $ExpectError
  const c3: MulticastChannel<void> = stdChannel()
  // $ExpectError
  const c4 = multicastChannel<void>()
  // $ExpectError
  const c5 = stdChannel<void>()
}
```

../redux-saga/packages/core/types/effects.d.ts

```
import { Last, Reverse } from 'typescript-tuple'

import {
  ActionPattern,
  Effect,
  Buffer,
  CombinatorEffect,
  CombinatorEffectDescriptor,
  SimpleEffect,
  END,
  Pattern,
  Predicate,
  Task,
```

```
  StrictEffect,
  ActionMatchingPattern,
} from '@redux-saga/types'

import { FlushableChannel, PuttableChannel, TakeableChannel, Action, AnyAction } from './index'

export { ActionPattern, Effect, Pattern, SimpleEffect, StrictEffect }

export const effectTypes: {
  TAKE: 'TAKE'
  PUT: 'PUT'
  ALL: 'ALL'
  RACE: 'RACE'
  CALL: 'CALL'
  CPS: 'CPS'
  FORK: 'FORK'
  JOIN: 'JOIN'
  CANCEL: 'CANCEL'
  SELECT: 'SELECT'
  ACTION_CHANNEL: 'ACTION_CHANNEL'
  CANCELLED: 'CANCELLED'
  FLUSH: 'FLUSH'
  GET_CONTEXT: 'GET_CONTEXT'
  SET_CONTEXT: 'SET_CONTEXT'
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
 * specified action on the Store. The Generator is suspended until an action
 * that matches `pattern` is dispatched.
 *
 * The result of `yield take(pattern)` is an action object being dispatched.
 *
 * `pattern` is interpreted using the following rules:
 *
 * - If `take` is called with no arguments or `'*'` all dispatched actions are
 *   matched (e.g. `take()` will match all actions)
 *
 * - If it is a function, the action is matched if `pattern(action)` is true
 *   (e.g. `take(action => action.entities)` will match all actions having a
 *   (truthy) `entities` field.)
 * > Note: if the pattern function has `toString` defined on it, `action.type`
 * > will be tested against `pattern.toString()` instead. This is useful if
 * > you're using an action creator library like redux-act or redux-actions.
 *
 * - If it is a String, the action is matched if `action.type === pattern` (e.g.
 *   `take(INCREMENT_ASYNC)`
 *
 * - If it is an array, each item in the array is matched with aforementioned
 *   rules, so the mixed array of strings and function predicates is supported.
 *   The most common use case is an array of strings though, so that
 *   `action.type` is matched against all items in the array (e.g.
 *   `take([INCREMENT, DECREMENT])` and that would match either actions of type
 *   `INCREMENT` or `DECREMENT`).
 *
 * The middleware provides a special action `END`. If you dispatch the END
 * action, then all Sagas blocked on a take Effect will be terminated regardless
 * of the specified pattern. If the terminated Saga has still some forked tasks
 * which are still running, it will wait for all the child tasks to terminate
 * before terminating the Task.
 */
export function take(pattern?: ActionPattern): TakeEffect
export function take<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

/**
 * Same as `take(pattern)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 *
 * #### Notes
 *
 * `takeMaybe` got its name from the FP analogy - it's like instead of having a
 * return type of `ACTION` (with automatic handling) we can have a type of
 * `Maybe(ACTION)` so we can handle both cases:
 *
 * - case when there is a `Just(ACTION)` (we have an action)
 * - the case of `NOTHING` (channel was closed*). i.e. we need some way to map
 *   over `END`
 *
 * internally all `dispatch`ed actions are going through the `stdChannel` which
 * is getting closed when `dispatch(END)` happens
 */
export function takeMaybe(pattern?: ActionPattern): TakeEffect
export function takeMaybe<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

export type TakeEffect = SimpleEffect<'TAKE', TakeEffectDescriptor>

export interface TakeEffectDescriptor {
  pattern: ActionPattern
  maybe?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
 * specified message from the provided Channel. If the channel is already
 * closed, then the Generator will immediately terminate following the same
 * process described above for `take(pattern)`.
 */
export function take<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

/**
 * Same as `take(channel)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 */
export function takeMaybe<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

export type ChannelTakeEffect<T> = SimpleEffect<'TAKE', ChannelTakeEffectDescriptor<T>>

export interface ChannelTakeEffectDescriptor<T> {
  channel: TakeableChannel<T>
  pattern?: Pattern<T>
```

```
  maybe?: boolean
}

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeEvery` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action:
 *
 *    import { takeEvery } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchFetchUser() {
 *      yield takeEvery('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeEvery` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeEvery = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * `takeEvery` allows concurrent actions to be handled. In the example above,
 * when a `USER_REQUESTED` action is dispatched, a new `fetchUser` task is
 * started even if a previous `fetchUser` is still pending (for example, the
 * user clicks on a `Load User` button 2 consecutive times at a rapid rate, the
 * 2nd click will dispatch a `USER_REQUESTED` action while the `fetchUser` fired
 * on the first one hasn't yet terminated)
 *
 * `takeEvery` doesn't handle out of order responses from tasks. There is no
 * guarantee that the tasks will terminate in the same order they were started.
 * To handle out of order responses, you may consider `takeLatest` below.
 *
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeEvery` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeEvery<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect
export function takeEvery<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect
export function takeEvery<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect
export function takeEvery<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeEvery(pattern, saga, ...args)`.
 */
export function takeEvery<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect
export function takeEvery<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. And automatically cancels any previous `saga` task started
 * previously if it's still running.
 *
 * Each time an action is dispatched to the store. And if this action matches
 * `pattern`, `takeLatest` starts a new `saga` task in the background. If a
 * `saga` task was started previously (on the last action dispatched before the
 * actual action), and if this task is still running, the task will be
 * cancelled.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLatest` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action. Since `takeLatest` cancels any pending task started
 * previously, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only conclude with the latest action
 *
 *    import { takeLatest } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLatest('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLatest` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
```

```
 *
 *    const takeLatest = (patternOrChannel, saga, ...args) => fork(function*() {
 *      let lastTask
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        if (lastTask) {
 *          yield cancel(lastTask) // cancel is no-op if the task has already terminated
 *        }
 *        lastTask = yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLatest` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeLatest<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect
export function takeLatest<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect
export function takeLatest<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect
export function takeLatest<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLatest(pattern, saga, ...args)`.
 */
export function takeLatest<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect
export function takeLatest<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. After spawning a task once, it blocks until spawned saga completes
 * and then starts to listen for a `pattern` again.
 *
 * In short, `takeLeading` is listening for the actions when it doesn't run a
 * saga.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLeading` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action. Since `takeLeading` ignores any new coming task
 * after it's started, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only keep on running with the leading
 * action
 *
 *    import { takeLeading } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLeading('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLeading` is a high-level API built using `take` and `call`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeLeading = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel);
 *        yield call(saga, ...args.concat(action));
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLeading` will
 *   add the incoming action to the argument list (i.e. the action will be the
 *   last argument provided to `saga`)
 */
export function takeLeading<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect
export function takeLeading<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect
export function takeLeading<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect
export function takeLeading<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLeading(pattern, saga, ...args)`.
 */
export function takeLeading<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect
export function takeLeading<T, Fn extends (...args: any[]) => any>(
```

```
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect

export type HelperWorkerParameters<T, Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends T
  ? AllButLast<Parameters<Fn>>
  : Parameters<Fn>

interface ThunkDispatch<State, ExtraThunkArg, BasicAction extends Action> {
  <ReturnType>(thunkAction: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>): ReturnType
  <Action extends BasicAction>(action: Action): Action
  <ReturnType, Action extends BasicAction>(
    action: Action | ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
  ): Action | ReturnType
}

export type ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction extends Action> = (
  dispatch: ThunkDispatch<State, ExtraThunkArg, BasicAction>,
  getState: () => State,
  extraArgument: ExtraThunkArg,
) => ReturnType

/**
 * Creates an Effect description that instructs the middleware to dispatch an
 * action to the Store. This effect is non-blocking, any errors that are
 * thrown downstream (e.g. in a reducer) will bubble back into the saga.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<A extends Action>(action: A): PutEffect<A>
export function put<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

/**
 * Just like `put` but the effect is blocking (if promise is returned from
 * `dispatch` it will wait for its resolution) and will bubble up errors from
 * downstream.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function putResolve<A extends Action>(action: A): PutEffect<A>
export function putResolve<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

export type PutEffect<A extends Action = AnyAction> = SimpleEffect<'PUT', PutEffectDescriptor<A>>

export interface PutEffectDescriptor<A extends Action> {
  action: A
  channel: null
  resolve?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to put an action
 * into the provided channel.
 *
 * This effect is blocking if the put is *not* buffered but immediately consumed
 * by takers. If an error is thrown in any of these takers it will bubble back
 * into the saga.
 *
 * @param channel a `Channel` Object.
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<T>(channel: PuttableChannel<T>, action: T | END): ChannelPutEffect<T>

export type ChannelPutEffect<T> = SimpleEffect<'PUT', ChannelPutEffectDescriptor<T>>

export interface ChannelPutEffectDescriptor<T> {
  action: T
  channel: PuttableChannel<T>
}

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments.
 *
 * #### Notes
 *
 * `fn` can be either a *normal* or a Generator function.
 *
 * The middleware invokes the function and examines its result.
 *
 * If the result is an Iterator object, the middleware will run that Generator
 * function, just like it did with the startup Generators (passed to the
 * middleware on startup). The parent Generator will be suspended until the
 * child Generator terminates normally, in which case the parent Generator is
 * resumed with the value returned by the child Generator. Or until the child
 * aborts with some error, in which case an error will be thrown inside the
 * parent Generator.
 *
 * If `fn` is a normal function and returns a Promise, the middleware will
 * suspend the Generator until the Promise is settled. After the promise is
 * resolved the Generator is resumed with the resolved value, or if the Promise
 * is rejected an error is thrown inside the Generator.
 *
 * If the result is not an Iterator object nor a Promise, the middleware will
 * immediately return that value back to the saga, so that it can resume its
 * execution synchronously.
 *
 * When an error is thrown inside the Generator, if it has a `try/catch` block
 * surrounding the current `yield` instruction, the control will be passed to
 * the `catch` block. Otherwise, the Generator aborts with the raised error, and
 * if this Generator was called by another Generator, the error will propagate
 * to the calling Generator.
 *
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
```

```
export function call<Fn extends (...args: any[]) => any>(fn: Fn, ...args: Parameters<Fn>): CallEffect
/**
 * Same as `call([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield call([localStorage, 'getItem'], 'redux-saga')`
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): CallEffect
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): CallEffect
/**
 * Same as `call(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): CallEffect
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): CallEffect

export type CallEffect = SimpleEffect<'CALL', CallEffectDescriptor>

export interface CallEffectDescriptor {
  context: any
  fn: Function
  args: any[]
}

/**
 * Alias for `call([context, fn], ...args)`.
 */
export function apply<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctx: Ctx,
  fnName: Name,
  args: Parameters<Ctx[Name]>,
): CallEffect
export function apply<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctx: Ctx,
  fn: Fn,
  args: Parameters<Fn>,
): CallEffect

/**
 * Creates an Effect description that instructs the middleware to invoke `fn` as
 * a Node style function.
 *
 * @param fn a Node style function. i.e. a function which accepts in addition to
 *   its arguments, an additional callback to be invoked by `fn` when it
 *   terminates. The callback accepts two parameters, where the first parameter
 *   is used to report errors while the second is used to report successful
 *   results
 * @param args an array to be passed as arguments for `fn`
 */
export function cps<Fn extends (cb: CpsCallback<any>) => any>(fn: Fn): CpsEffect
export function cps<Fn extends (...args: any[]) => any>(fn: Fn, ...args: CpsFunctionParameters<Fn>): CpsEffect
/**
 * Same as `cps([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield cps([localStorage, 'getItem'], 'redux-saga')`
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: CpsFunctionParameters<Ctx[Name]>
): CpsEffect
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: CpsFunctionParameters<Ctx[Name]>
): CpsEffect
/**
 * Same as `cps(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: [Ctx, Fn],
  ...args: CpsFunctionParameters<Fn>
): CpsEffect
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: CpsFunctionParameters<Fn>
): CpsEffect
```

```
export type CpsFunctionParameters<Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends CpsCallback<any>
  ? AllButLast<Parameters<Fn>>
  : never

export interface CpsCallback<R> {
  (error: any, result: R): void
  cancel?(): void
}

export type CpsEffect = SimpleEffect<'CPS', CallEffectDescriptor>

/**
 * Creates an Effect description that instructs the middleware to perform a
 * *non-blocking call* on `fn`
 *
 * returns a `Task` object.
 *
 * #### Note
 *
 * `fork`, like `call`, can be used to invoke both normal and Generator
 * functions. But, the calls are non-blocking, the middleware doesn't suspend
 * the Generator while waiting for the result of `fn`. Instead as soon as `fn`
 * is invoked, the Generator resumes immediately.
 *
 * `fork`, alongside `race`, is a central Effect for managing concurrency
 * between Sagas.
 *
 * The result of `yield fork(fn ...args)` is a `Task` object.  An object
 * with some useful methods and properties.
 *
 * All forked tasks are *attached* to their parents. When the parent terminates
 * the execution of its own body of instructions, it will wait for all forked
 * tasks to terminate before returning.
 *
 * Errors from child tasks automatically bubble up to their parents. If any
 * forked task raises an uncaught error, then the parent task will abort with
 * the child Error, and the whole Parent's execution tree (i.e. forked tasks +
 * the *main task* represented by the parent's body if it's still running) will
 * be cancelled.
 *
 * Cancellation of a forked Task will automatically cancel all forked tasks that
 * are still executing. It'll also cancel the current Effect where the cancelled
 * task was blocked (if any).
 *
 * If a forked task fails *synchronously* (ie: fails immediately after its
 * execution before performing any async operation), then no Task is returned,
 * instead the parent will be aborted as soon as possible (since both parent and
 * child execute in parallel, the parent will abort as soon as it takes notice
 * of the child failure).
 *
 * To create *detached* forks, use `spawn` instead.
 *
 * @param fn A Generator function, or normal function which returns a Promise as result
 * @param args An array of values to be passed as arguments to `fn`
 */
export function fork<Fn extends (...args: any[]) => any>(fn: Fn, ...args: Parameters<Fn>): ForkEffect
/**
 * Same as `fork([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield fork([localStorage, 'getItem'], 'redux-saga')`
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect
/**
 * Same as `fork(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect

export type ForkEffect = SimpleEffect<'FORK', ForkEffectDescriptor>

export interface ForkEffectDescriptor extends CallEffectDescriptor {
  detached?: boolean
}

/**
 * Same as `fork(fn, ...args)` but creates a *detached* task. A detached task
 * remains independent from its parent and acts like a top-level task. The
 * parent will not wait for detached tasks to terminate before returning and all
 * events which may affect the parent or the detached task are completely
 * independents (error, cancellation).
 */
export function spawn<Fn extends (...args: any[]) => any>(fn: Fn, ...args: Parameters<Fn>): ForkEffect
/**
 * Same as `spawn([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield spawn([localStorage, 'getItem'], 'redux-saga')`
```

```
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect
/**
 * Same as `spawn(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect

/**
 * Creates an Effect description that instructs the middleware to wait for the
 * result of a previously forked task.
 *
 * #### Notes
 *
 * `join` will resolve to the same outcome of the joined task (success or
 * error). If the joined task is cancelled, the cancellation will also propagate
 * to the Saga executing the join effect. Similarly, any potential callers of
 * those joiners will be cancelled as well.
 *
 * @param task A `Task` object returned by a previous `fork`
 */
export function join(task: Task): JoinEffect
/**
 * Creates an Effect description that instructs the middleware to wait for the
 * results of previously forked tasks.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function join(tasks: Task[]): JoinEffect

export type JoinEffect = SimpleEffect<'JOIN', JoinEffectDescriptor>

export type JoinEffectDescriptor = Task | Task[]

/**
 * Creates an Effect description that instructs the middleware to cancel a
 * previously forked task.
 *
 * #### Notes
 *
 * To cancel a running task, the middleware will invoke `return` on the
 * underlying Generator object. This will cancel the current Effect in the task
 * and jump to the finally block (if defined).
 *
 * Inside the finally block, you can execute any cleanup logic or dispatch some
 * action to keep the store in a consistent state (e.g. reset the state of a
 * spinner to false when an ajax request is cancelled). You can check inside the
 * finally block if a Saga was cancelled by issuing a `yield cancelled()`.
 *
 * Cancellation propagates downward to child sagas. When cancelling a task, the
 * middleware will also cancel the current Effect (where the task is currently
 * blocked). If the current Effect is a call to another Saga, it will be also
 * cancelled. When cancelling a Saga, all *attached forks* (sagas forked using
 * `yield fork()`) will be cancelled. This means that cancellation effectively
 * affects the whole execution tree that belongs to the cancelled task.
 *
 * `cancel` is a non-blocking Effect. i.e. the Saga executing it will resume
 * immediately after performing the cancellation.
 *
 * For functions which return Promise results, you can plug your own
 * cancellation logic by attaching a `[CANCEL]` to the promise.
 *
 * The following example shows how to attach cancellation logic to a Promise
 * result:
 *
 *    import { CANCEL } from 'redux-saga'
 *    import { fork, cancel } from 'redux-saga/effects'
 *
 *    function myApi() {
 *      const promise = myXhr(...)
 *
 *      promise[CANCEL] = () => myXhr.abort()
 *      return promise
 *    }
 *
 *    function* mySaga() {
 *
 *      const task = yield fork(myApi)
 *
 *      // ... later
 *      // will call promise[CANCEL] on the result of myApi
 *      yield cancel(task)
 *    }
 *
 * redux-saga will automatically cancel jqXHR objects using their `abort` method.
 *
```

```
 * @param task A `Task` object returned by a previous `fork`
 */
export function cancel(task: Task): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel
 * previously forked tasks.
 *
 * #### Notes
 *
 * It wraps the array of tasks in cancel effects, roughly becoming the
 * equivalent of `yield tasks.map(t => cancel(t))`.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function cancel(tasks: Task[]): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel a task
 * in which it has been yielded (self-cancellation). It allows to reuse
 * destructor-like logic inside a `finally` blocks for both outer
 * (`cancel(task)`) and self (`cancel()`) cancellations.
 *
 * #### Example
 *
 *    function* deleteRecord({ payload }) {
 *      try {
 *        const { confirm, deny } = yield call(prompt);
 *        if (confirm) {
 *          yield put(actions.deleteRecord.confirmed())
 *        }
 *        if (deny) {
 *          yield cancel()
 *        }
 *      } catch(e) {
 *        // handle failure
 *      } finally {
 *        if (yield cancelled()) {
 *          // shared cancellation logic
 *          yield put(actions.deleteRecord.cancel(payload))
 *        }
 *      }
 *    }
 */
export function cancel(): CancelEffect

export type CancelEffect = SimpleEffect<'CANCEL', CancelEffectDescriptor>

export type CancelEffectDescriptor = Task | Task[] | SELF_CANCELLATION
type SELF_CANCELLATION = '@@redux-saga/SELF_CANCELLATION'

/**
 * Creates an effect that instructs the middleware to invoke the provided
 * selector on the current Store's state (i.e. returns the result of
 * `selector(getState(), ...args)`).
 *
 * If `select` is called without argument (i.e. `yield select()`) then the
 * effect is resolved with the entire state (the same result of a `getState()`
 * call).
 *
 * > It's important to note that when an action is dispatched to the store, the
 * middleware first forwards the action to the reducers and then notifies the
 * Sagas. This means that when you query the Store's State, you get the State
 * **after** the action has been applied. However, this behavior is only
 * guaranteed if all subsequent middlewares call `next(action)` synchronously.
 * If any subsequent middleware calls `next(action)` asynchronously (which is
 * unusual but possible), then the sagas will get the state from **before** the
 * action is applied.  Therefore it is recommended to review the source of each
 * subsequent middleware to ensure it calls `next(action)` synchronously, or
 * else ensure that redux-saga is the last middleware in the call chain.
 *
 * #### Notes
 *
 * Preferably, a Saga should be autonomous and should not depend on the Store's
 * state. This makes it easy to modify the state implementation without
 * affecting the Saga code. A saga should preferably depend only on its own
 * internal control state when possible. But sometimes, one could find it more
 * convenient for a Saga to query the state instead of maintaining the needed
 * data by itself (for example, when a Saga duplicates the logic of invoking
 * some reducer to compute a state that was already computed by the Store).
 *
 * For example, suppose we have this state shape in our application:
 *
 *    state = {
 *      cart: {...}
 *    }
 *
 * We can create a *selector*, i.e. a function which knows how to extract the
 * `cart` data from the State:
 *
 * `./selectors`
 *
 *    export const getCart = state => state.cart
 *
 *
 * Then we can use that selector from inside a Saga using the `select` Effect:
 *
 * `./sagas.js`
 *
 *    import { take, fork, select } from 'redux-saga/effects'
 *    import { getCart } from './selectors'
 *
 *    function* checkout() {
 *      // query the state using the exported selector
 *      const cart = yield select(getCart)
 *
 *      // ... call some API endpoint then dispatch a success/error action
 *    }
 *
 *    export default function* rootSaga() {
 *      while (true) {
 *        yield take('CHECKOUT_REQUEST')
 *        yield fork(checkout)
 *      }
```

```
 *    }
 *
 * `checkout` can get the needed information directly by using
 * `select(getCart)`. The Saga is coupled only with the `getCart` selector. If
 * we have many Sagas (or React Components) that needs to access the `cart`
 * slice, they will all be coupled to the same function `getCart`. And if we now
 * change the state shape, we need only to update `getCart`.
 *
 * @param selector a function `(state, ...args) => args`. It takes the current
 *   state and optionally some arguments and returns a slice of the current
 *   Store's state
 * @param args optional arguments to be passed to the selector in addition of
 *   `getState`.
 */
export function select(): SelectEffect
export function select<Fn extends (state: any, ...args: any[]) => any>(
  selector: Fn,
  ...args: Tail<Parameters<Fn>>
): SelectEffect

export type SelectEffect = SimpleEffect<'SELECT', SelectEffectDescriptor>

export interface SelectEffectDescriptor {
  selector(state: any, ...args: any[]): any
  args: any[]
}

/**
 * Creates an effect that instructs the middleware to queue the actions matching
 * `pattern` using an event channel. Optionally, you can provide a buffer to
 * control buffering of the queued actions.
 *
 * #### Example
 *
 * The following code creates a channel to buffer all `USER_REQUEST` actions.
 * Note that even the Saga may be blocked on the `call` effect. All actions that
 * come while it's blocked are automatically buffered. This causes the Saga to
 * execute the API calls one at a time
 *
 *    import { actionChannel, call } from 'redux-saga/effects'
 *    import api from '...'
 *
 *    function* takeOneAtMost() {
 *      const chan = yield actionChannel('USER_REQUEST')
 *      while (true) {
 *        const {payload} = yield take(chan)
 *        yield call(api.getUser, payload)
 *      }
 *    }
 *
 * @param pattern see API for `take(pattern)`
 * @param buffer a `Buffer` object
 */
export function actionChannel(pattern: ActionPattern, buffer?: Buffer<Action>): ActionChannelEffect

export type ActionChannelEffect = SimpleEffect<'ACTION_CHANNEL', ActionChannelEffectDescriptor>

export interface ActionChannelEffectDescriptor {
  pattern: ActionPattern
  buffer?: Buffer<Action>
}

/**
 * Creates an effect that instructs the middleware to flush all buffered items
 * from the channel. Flushed items are returned back to the saga, so they can be
 * utilized if needed.
 *
 * #### Example
 *
 *    function* saga() {
 *      const chan = yield actionChannel('ACTION')
 *
 *      try {
 *        while (true) {
 *          const action = yield take(chan)
 *          // ...
 *        }
 *      } finally {
 *        const actions = yield flush(chan)
 *        // ...
 *      }
 *    }
 *
 * @param channel a `Channel` Object.
 */
export function flush<T>(channel: FlushableChannel<T>): FlushEffect<T>

export type FlushEffect<T> = SimpleEffect<'FLUSH', FlushEffectDescriptor<T>>

export type FlushEffectDescriptor<T> = FlushableChannel<T>

/**
 * Creates an effect that instructs the middleware to return whether this
 * generator has been cancelled. Typically you use this Effect in a finally
 * block to run Cancellation specific code
 *
 * #### Example
 *
 *    function* saga() {
 *      try {
 *        // ...
 *      } finally {
 *        if (yield cancelled()) {
 *          // logic that should execute only on Cancellation
 *        }
 *        // logic that should execute in all situations (e.g. closing a channel)
 *      }
 *    }
 */
export function cancelled(): CancelledEffect

export type CancelledEffect = SimpleEffect<'CANCELLED', CancelledEffectDescriptor>
```

```
export type CancelledEffectDescriptor = {}

/**
 * Creates an effect that instructs the middleware to update its own context.
 * This effect extends saga's context instead of replacing it.
 */
export function setContext<C extends object>(props: C): SetContextEffect<C>

export type SetContextEffect<C extends object> = SimpleEffect<'SET_CONTEXT', SetContextEffectDescriptor<C>>

export type SetContextEffectDescriptor<C extends object> = C

/**
 * Creates an effect that instructs the middleware to return a specific property
 * of saga's context.
 */
export function getContext(prop: string): GetContextEffect

export type GetContextEffect = SimpleEffect<'GET_CONTEXT', GetContextEffectDescriptor>

export type GetContextEffectDescriptor = string

/**
 * Returns an effect descriptor to block execution for `ms` milliseconds and return `val` value.
 */
export function delay<T = true>(ms: number, val?: T): CallEffect

/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * After spawning a task it's still accepting incoming actions into the
 * underlying `buffer`, keeping at most 1 (the most recent one), but in the same
 * time holding up with spawning new task for `ms` milliseconds (hence its name -
 * `throttle`). Purpose of this is to ignore incoming actions for a given
 * period of time while processing a task.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `throttle` to start a new `fetchAutocomplete` task on dispatched
 * `FETCH_AUTOCOMPLETE` action. However since `throttle` ignores consecutive
 * `FETCH_AUTOCOMPLETE` for some time, we ensure that user won't flood our
 * server with requests.
 *
 *    import { call, put, throttle } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* throttleAutocomplete() {
 *      yield throttle(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `throttle` is a high-level API built using `take`, `fork` and
 * `actionChannel`. Here is how the helper could be implemented using the
 * low-level Effects
 *
 *    const throttle = (ms, pattern, task, ...args) => fork(function*() {
 *      const throttleChannel = yield actionChannel(pattern, buffers.sliding(1))
 *
 *      while (true) {
 *        const action = yield take(throttleChannel)
 *        yield fork(task, ...args, action)
 *        yield delay(ms)
 *      }
 *    })
 *
 * @param ms length of a time window in milliseconds during which actions will
 *   be ignored after the action starts processing
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `throttle` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function throttle<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect
export function throttle<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect
export function throttle<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect
export function throttle<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `throttle(ms, pattern, saga, ...args)`.
 */
export function throttle<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect
export function throttle<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
```

```
): ForkEffect

/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * Saga will be called after it stops taking `pattern` actions for `ms`
 * milliseconds. Purpose of this is to prevent calling saga until the actions
 * are settled off.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `debounce` to delay calling `fetchAutocomplete` saga until we stop receive
 * any `FETCH_AUTOCOMPLETE` events for at least `1000` ms.
 *
 *    import { call, put, debounce } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* debounceAutocomplete() {
 *      yield debounce(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `debounce` is a high-level API built using `take`, `delay` and `fork`. Here
 * is how the helper could be implemented using the low-level Effects
 *
 *    const debounce = (ms, pattern, task, ...args) => fork(function*() {
 *      while (true) {
 *        let action = yield take(pattern)
 *
 *        while (true) {
 *          const { debounced, _action } = yield race({
 *            debounced: delay(ms),
 *            _action: take(pattern)
 *          })
 *
 *          if (debounced) {
 *            yield fork(worker, ...args, action)
 *            break
 *          }
 *
 *          action = _action
 *        }
 *      }
 *    })
 *
 * @param ms defines how many milliseconds should elapse since the last time
 *   `pattern` action was fired to call the `saga`
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `debounce` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function debounce<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect
export function debounce<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect
export function debounce<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect
export function debounce<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `debounce(ms, pattern, saga, ...args)`.
 */
export function debounce<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect
export function debounce<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments. In case of failure will try to make
 * another call after `delay` milliseconds, if a number of attempts < `maxTries`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `retrySaga`. We use `retry`
 * to try to fetch our API 3 times with 10 second interval. If `request` fails
 * first time than `retry` will call `request` one more time while calls count
 * less than 3.
 *
 *    import { put, retry } from 'redux-saga/effects'
 *    import { request } from 'some-api';
 *
 *    function* retrySaga(data) {
 *      try {
 *        const SECOND = 1000
 *        const response = yield retry(3, 10 * SECOND, request, data)
```

```
 *        yield put({ type: 'REQUEST_SUCCESS', payload: response })
 *      } catch(error) {
 *        yield put({ type: 'REQUEST_FAIL', payload: { error } })
 *      }
 *    }
 *
 * @param maxTries maximum calls count.
 * @param delay length of a time window in milliseconds between `fn` calls.
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as a result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
export function retry<Fn extends (...args: any[]) => any>(
  maxTries: number,
  delayLength: number,
  fn: Fn,
  ...args: Parameters<Fn>
): CallEffect

/**
 * Creates an Effect description that instructs the middleware to run multiple
 * Effects in parallel and wait for all of them to complete. It's quite the
 * corresponding API to standard
 * [`Promise#all`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/all).
 *
 * #### Example
 *
 * The following example runs two blocking calls in parallel:
 *
 *    import { fetchCustomers, fetchProducts } from './path/to/api'
 *    import { all, call } from `redux-saga/effects`
 *
 *    function* mySaga() {
 *      const [customers, products] = yield all([
 *        call(fetchCustomers),
 *        call(fetchProducts)
 *      ])
 *    }
 */
export function all<T>(effects: T[]): AllEffect<T>
/**
 * The same as `all([...effects])` but let's you to pass in a dictionary object
 * of effects with labels, just like `race(effects)`
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function all<T>(effects: { [key: string]: T }): AllEffect<T>

export type AllEffect<T> = CombinatorEffect<'ALL', T>

export type AllEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * Creates an Effect description that instructs the middleware to run a *Race*
 * between multiple Effects (this is similar to how
 * [`Promise.race([...])`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)
 * behaves).
 *
 *
 * #### Example
 *
 * The following example runs a race between two effects:
 *
 * 1. A call to a function `fetchUsers` which returns a Promise
 * 2. A `CANCEL_FETCH` action which may be eventually dispatched on the Store
 *
 *
 *    import { take, call, race } from `redux-saga/effects`
 *    import fetchUsers from './path/to/fetchUsers'
 *
 *    function* fetchUsersSaga() {
 *      const { response, cancel } = yield race({
 *        response: call(fetchUsers),
 *        cancel: take(CANCEL_FETCH)
 *      })
 *    }
 *
 * If `call(fetchUsers)` resolves (or rejects) first, the result of `race` will
 * be an object with a single keyed object `{response: result}` where `result`
 * is the resolved result of `fetchUsers`.
 *
 * If an action of type `CANCEL_FETCH` is dispatched on the Store before
 * `fetchUsers` completes, the result will be a single keyed object
 * `{cancel: action}`, where action is the dispatched action.
 *
 * #### Notes
 *
 * When resolving a `race`, the middleware automatically cancels all the losing
 * Effects.
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function race<T>(effects: { [key: string]: T }): RaceEffect<T>
/**
 * The same as `race(effects)` but lets you pass in an array of effects.
 */
export function race<T>(effects: T[]): RaceEffect<T>

export type RaceEffect<T> = CombinatorEffect<'RACE', T>

export type RaceEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * [H, ...T] -> T
 */
export type Tail<L extends any[]> = ((...l: L) => any) extends (h: any, ...t: infer T) => any ? T : never
/**
 * [...A, B] -> A
 */
export type AllButLast<L extends any[]> = Reverse<Tail<Reverse<L>>>
```

../redux-saga/packages/core/types/effects.test.ts

```
import { SagaIterator, Channel, EventChannel, MulticastChannel, Task, Buffer, END, buffers, detach } from 'redux-saga'
import {
  take,
  takeMaybe,
  put,
  putResolve,
  call,
  apply,
  cps,
  fork,
  spawn,
  join,
  cancel,
  select,
  actionChannel,
  cancelled,
  flush,
  setContext,
  getContext,
  takeEvery,
  takeLatest,
  takeLeading,
  throttle,
  delay,
  retry,
  all,
  race,
  debounce,
} from 'redux-saga/effects'
import { Action, ActionCreator } from 'redux'
import { StringableActionCreator, ActionMatchingPattern } from '@redux-saga/types'
import { ThunkAction } from '@redux-saga/core/effects'

interface MyAction extends Action {
  customField: string
}

declare const stringableActionCreator: ActionCreator<MyAction>

Object.assign(stringableActionCreator, {
  toString() {
    return 'my-action'
  },
})

const isMyAction = (action: Action): action is MyAction => {
  return action.type === 'my-action'
}

interface ChannelItem {
  someField: string
}
declare const channel: Channel<ChannelItem>
declare const eventChannel: EventChannel<ChannelItem>
declare const multicastChannel: MulticastChannel<ChannelItem>

function* testTake(): SagaIterator {
  yield take()
  yield take('my-action')
  yield take((action: Action) => action.type === 'my-action')
  yield take(isMyAction)

  // $ExpectError
  yield take(() => {})

  yield take(stringableActionCreator)

  yield take(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  // $ExpectError
  yield take([() => {}])

  yield takeMaybe(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  yield take(channel)
  yield takeMaybe(channel)

  yield take(eventChannel)
  yield takeMaybe(eventChannel)

  yield take(multicastChannel)
  yield takeMaybe(multicastChannel)

  // $ExpectError
  yield take(multicastChannel, (input: { someField: number }) => input.someField === 'foo')
  yield take(multicastChannel, (input: ChannelItem) => input.someField === 'foo')

  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield take([pattern1, pattern2])
  yield takeMaybe([pattern1, pattern2])
}

function* testPut(): SagaIterator {
  type TestThunkAction = ThunkAction<number, object, object, { type: 'thunk-action' }>
  const thunkActionCreator = (): TestThunkAction => (dispatch) => {
    dispatch({ type: 'thunk-action' })

    return 42
  }

  yield put({ type: 'my-action' })
  yield put(thunkActionCreator())

  // $ExpectError
  yield put(channel, { type: 'my-action' })
```

```
  yield put(channel, { someField: '--' })
  yield put(channel, END)

  // $ExpectError
  yield put(eventChannel, { someField: '--' })
  // $ExpectError
  yield put(eventChannel, END)

  yield put(multicastChannel, { someField: '--' })
  yield put(multicastChannel, END)

  yield putResolve({ type: 'my-action' })
  yield putResolve(thunkActionCreator())
}

function* testCall(): SagaIterator {
  // $ExpectError
  yield call()

  // $ExpectError
  yield call({})

  yield call(() => {})

  // $ExpectError
  yield call((a: 'a') => {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // yield call(function*(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  yield call((a: 'a') => {}, 1)
  // $ExpectError
  yield call(function*(a: 'a'): SagaIterator {}, 1)
  yield call((a: 'a') => {}, 'a')
  yield call(function*(a: 'a'): SagaIterator {}, 'a')

  yield call<(a: 'a') => number>((a: 'a') => 1, 'a')

  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 1, 'b')
  yield call((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield call<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => number>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => 1,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
    'g',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: 'bar') {
      return this.foo
    },
  }

  // $ExpectError
  yield call([obj, obj.foo])
  // $ExpectError
  yield call([obj, obj.getFoo])
  yield call([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield call([obj, obj.getFoo], 1)

  // $ExpectError
  yield call([obj, 'foo'])
  // $ExpectError
  yield call([obj, 'getFoo'])
  // $ExpectError
  yield call([obj, 'getFoo'], 1)
  yield call([obj, 'getFoo'], 'bar')
  yield call<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')

  // $ExpectError
  yield call({ context: obj, fn: obj.foo })
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo })
  yield call({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield call({ context: obj, fn: 'foo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' }, 1)
  yield call({ context: obj, fn: 'getFoo' }, 'bar')
  yield call<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
}

function* testApply(): SagaIterator {
  const obj = {
    foo: 'bar',
    getFoo() {
```

```
    return this.foo
  },
  meth1(a: string) {
    return 1
  },
  meth2(a: string, b: number) {
    return 1
  },
  meth7(a: string, b: number, c: string, d: number, e: string, f: number, g: string) {
    return 1
  },
}

// $ExpectError
yield apply(obj, obj.foo, [])
yield apply(obj, obj.getFoo, [])
yield apply<typeof obj, () => string>(obj, obj.getFoo, [])

// $ExpectError
yield apply(obj, 'foo', [])
yield apply(obj, 'getFoo', [])
yield apply<typeof obj, 'getFoo'>(obj, 'getFoo', [])

// $ExpectError
yield apply(obj, obj.meth1)
// $ExpectError
yield apply(obj, obj.meth1, [])
// $ExpectError
yield apply(obj, obj.meth1, [1])
yield apply(obj, obj.meth1, ['a'])
yield apply<typeof obj, (a: string) => number>(obj, obj.meth1, ['a'])

// $ExpectError
yield apply(obj, 'meth1')
// $ExpectError
yield apply(obj, 'meth1', [])
// $ExpectError
yield apply(obj, 'meth1', [1])
yield apply(obj, 'meth1', ['a'])
yield apply<typeof obj, 'meth1'>(obj, 'meth1', ['a'])

// $ExpectError
yield apply(obj, obj.meth2, ['a'])
// $ExpectError
yield apply(obj, obj.meth2, ['a', 'b'])
// $ExpectError
yield apply(obj, obj.meth2, [1, 'b'])
yield apply(obj, obj.meth2, ['a', 1])
yield apply<typeof obj, (a: string, b: number) => number>(obj, obj.meth2, ['a', 1])

// $ExpectError
yield apply(obj, 'meth2', ['a'])
// $ExpectError
yield apply(obj, 'meth2', ['a', 'b'])
// $ExpectError
yield apply(obj, 'meth2', [1, 'b'])
yield apply(obj, 'meth2', ['a', 1])
yield apply<typeof obj, 'meth2'>(obj, 'meth2', ['a', 1])

// $ExpectError
yield apply(obj, obj.meth7, [1, 'b', 'c', 'd', 'e', 'f', 'g'])
yield apply(obj, obj.meth7, ['a', 1, 'b', 2, 'c', 3, 'd'])
yield apply<typeof obj, (a: string, b: number, c: string, d: number, e: string, f: number, g: string) => number>(
  obj,
  obj.meth7,
  ['a', 1, 'b', 2, 'c', 3, 'd'],
)

// $ExpectError
yield apply(obj, 'meth7', [1, 'b', 'c', 'd', 'e', 'f', 'g'])
yield apply(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
yield apply<typeof obj, 'meth7'>(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
}

function* testCps(): SagaIterator {
  type Cb<R> = (error: any, result: R) => void

  // $ExpectError
  yield cps((a: number) => {})

  // $ExpectError
  yield cps((a: number, b: string) => {}, 42)

  yield cps(cb => {
    cb(null, 1)
  })
  yield cps((cb: Cb<number>) => {
    cb(null, 1)
  })

  yield cps<(cb: Cb<string>) => void>(cb => {
    cb(null, 1)  // $ExpectError
  })
  yield cps<(cb: Cb<number>) => void>(cb => {
    cb(null, 1)
  })

  yield cps(cb => {
    cb.cancel = () => {}
  })

  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {})
  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {}, 1)
  yield cps((a: 'a', cb: Cb<number>) => {}, 'a')

  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a')
  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a', 1)
  // $ExpectError
```

```
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 1, 'b')
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 'a', 'b')

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {}, 1, 'b', 'c', 'd')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {}, 1, 'b', 'c', 'd', 'e', 'f')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: string, cb: Cb<string>) {
      cb(null, this.foo)
    },
  }
  const objWithoutCb = {
    foo: 'bar',
    getFoo(arg: string) {},
  }

  // $ExpectError
  yield cps([obj, obj.foo])
  // $ExpectError
  yield cps([obj, obj.getFoo])
  // $ExpectError
  yield cps([obj, obj.getFoo], 1)
  yield cps([obj, obj.getFoo], 'bar')
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield cps([objWithoutCb, objWithoutCb.getFoo])

  // $ExpectError
  yield cps([obj, 'foo'])
  // $ExpectError
  yield cps([obj, 'getFoo'])
  // $ExpectError
  yield cps([obj, 'getFoo'], 1)
  yield cps([obj, 'getFoo'], 'bar')
  yield cps<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield cps([objWithoutCb, 'getFoo'])

  // $ExpectError
  yield cps({ context: obj, fn: obj.foo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo }, 1)
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: objWithoutCb.getFoo })

  // $ExpectError
  yield cps({ context: obj, fn: 'foo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' }, 1)
  yield cps({ context: obj, fn: 'getFoo' }, 'bar')
  yield cps<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: 'getFoo' })
}

function* testFork(): SagaIterator {
  // $ExpectError
  yield fork()

  yield fork(() => {})
```

```
  // $ExpectError
  yield fork((a: 'a') => {})
  // $ExpectError
  yield fork((a: 'a') => {}, 1)
  yield fork((a: 'a') => {}, 'a')

  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 1, 'b')
  yield fork((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield fork([obj, obj.foo])
  // $ExpectError
  yield fork([obj, obj.getFoo])
  yield fork([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield fork([obj, obj.getFoo], 1)

  // $ExpectError
  yield fork([obj, 'foo'])
  // $ExpectError
  yield fork([obj, 'getFoo'])
  yield fork([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield fork([obj, 'getFoo'], 1)

  // $ExpectError
  yield fork({ context: obj, fn: obj.foo })
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo })
  yield fork({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield fork({ context: obj, fn: 'foo' })
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' })
  yield fork({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' }, 1)
}

function* testSpawn(): SagaIterator {
  // $ExpectError
  yield spawn()

  yield spawn(() => {})

  // $ExpectError
  yield spawn((a: 'a') => {})
  // $ExpectError
  yield spawn((a: 'a') => {}, 1)
  yield spawn((a: 'a') => {}, 'a')

  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 1, 'b')
  yield spawn((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield spawn([obj, obj.foo])
  // $ExpectError
  yield spawn([obj, obj.getFoo])
  yield spawn([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield spawn([obj, obj.getFoo], 1)

  // $ExpectError
  yield spawn([obj, 'foo'])
  // $ExpectError
  yield spawn([obj, 'getFoo'])
  yield spawn([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield spawn([obj, 'getFoo'], 1)

  // $ExpectError
  yield spawn({ context: obj, fn: obj.foo })
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo })
  yield spawn({ context: obj, fn: obj.getFoo }, 'bar')
```

```
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield spawn({ context: obj, fn: 'foo' })
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' })
  yield spawn({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' }, 1)
}

declare const task: Task

function* testJoin(): SagaIterator {
  // $ExpectError
  yield join()

  // $ExpectError
  yield join({})

  yield join(task)
  // $ExpectError
  yield join(task, task)
  yield join([task, task])
  yield join([task, task, task])

  // $ExpectError
  yield join([task, task, {}])
}

function* testCancel(): SagaIterator {
  yield cancel()

  // $ExpectError
  yield cancel(undefined)
  // $ExpectError
  yield cancel({})

  yield cancel(task)
  // $ExpectError
  yield cancel(task, task)
  yield cancel([task, task])
  yield cancel([task, task, task])

  const tasks: Task[] = []

  yield cancel(tasks)

  // $ExpectError
  yield cancel([task, task, {}])
}

function* testDetach(): SagaIterator {
  yield detach(fork(() => {}))

  // $ExpectError
  yield detach(call(() => {}))
}

function* testSelect(): SagaIterator {
  interface State {
    foo: string
  }

  yield select()

  yield select((state: State) => state.foo)
  // $ExpectError
  yield select<(state: State) => number>((state: State) => state.foo)
  yield select<(state: State) => string>((state: State) => state.foo)

  // $ExpectError
  yield select((state: State, a: 'a') => state.foo)
  // $ExpectError
  yield select((state: State, a: 'a') => state.foo, 1)
  yield select((state: State, a: 'a') => state.foo, 'a')
  yield select<(state: State, a: 'a') => string>((state: State, a: 'a') => state.foo, 'a')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a')
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 1)
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 1, 'b')
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')
  yield select<(state: State, a: 'a', b: 'b') => string>((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo, 1, 'b', 'c', 'd', 'e', 'f')

  yield select(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
  yield select<(state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => string>(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
}

declare const actionBuffer: Buffer<Action>
```

```
declare const nonActionBuffer: Buffer<ChannelItem>

function* testActionChannel(): SagaIterator {
  // $ExpectError
  yield actionChannel()

  /* action type */

  yield actionChannel('my-action')
  yield actionChannel('my-action', actionBuffer)
  // $ExpectError
  yield actionChannel('my-action', nonActionBuffer)

  /* action predicate */

  yield actionChannel((action: Action) => action.type === 'my-action')
  yield actionChannel((action: Action) => action.type === 'my-action', actionBuffer)
  // $ExpectError
  yield actionChannel((action: Action) => action.type === 'my-action', nonActionBuffer)
  // $ExpectError
  yield actionChannel((item: ChannelItem) => item.someField === '--', actionBuffer)

  // $ExpectError
  yield actionChannel(() => {})
  // $ExpectError
  yield actionChannel(() => {}, actionBuffer)

  /* stringable action creator */

  yield actionChannel(stringableActionCreator)

  yield actionChannel(stringableActionCreator, buffers.fixed<MyAction>())
  // $ExpectError
  yield actionChannel(stringableActionCreator, nonActionBuffer)

  /* array */

  yield actionChannel(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator])

  // $ExpectError
  yield actionChannel([() => {}])
}

function* testCancelled(): SagaIterator {
  yield cancelled()
  // $ExpectError
  yield cancelled(1)
}

function* testFlush(): SagaIterator {
  // $ExpectError
  yield flush()
  // $ExpectError
  yield flush({})

  yield flush(channel)
  yield flush(eventChannel)
  // $ExpectError
  yield flush(multicastChannel)
}

function* testGetContext(): SagaIterator {
  // $ExpectError
  yield getContext()

  // $ExpectError
  yield getContext({})

  yield getContext('prop')
}

function* testSetContext(): SagaIterator {
  // $ExpectError
  yield setContext()

  // $ExpectError
  yield setContext('prop')

  yield setContext({ prop: 1 })
}

function* testTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery()
  // $ExpectError
  yield takeEvery('my-action')

  yield takeEvery('my-action', (action: Action) => {})
  yield takeEvery('my-action', (action: MyAction) => {})
  yield takeEvery('my-action', function*(action: Action): SagaIterator {})
  yield takeEvery('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker1)
  // $ExpectError
  yield takeEvery('my-action', helperWorker1, 1)
  yield takeEvery('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga1)
  // $ExpectError
  yield takeEvery('my-action', helperSaga1, 1)
  yield takeEvery('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
```

```
  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const helperWorker8 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeEvery(isMyAction, action => action.customField)

  yield takeEvery(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeEvery(() => {}, (action: Action) => {})

  yield takeEvery(stringableActionCreator, action => action.customField)

  yield takeEvery(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeEvery(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeEvery([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeEvery(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery(channel)

  // $ExpectError
  yield takeEvery(channel, (action: Action) => {})
  yield takeEvery(channel, (action: ChannelItem) => {})
  yield takeEvery(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeEvery(channel, helperWorker1)
  // $ExpectError
  yield takeEvery(channel, helperWorker1, 1)
  yield takeEvery(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga1)
  // $ExpectError
  yield takeEvery(channel, helperSaga1, 1)
  yield takeEvery(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}
```

```
  // $ExpectError
  yield takeEvery(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery(eventChannel, (action: ChannelItem) => {})
  yield takeEvery(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest()
  // $ExpectError
  yield takeLatest('my-action')

  yield takeLatest('my-action', (action: Action) => {})
  yield takeLatest('my-action', (action: MyAction) => {})
  yield takeLatest('my-action', function*(action: Action): SagaIterator {})
  yield takeLatest('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker1)
  // $ExpectError
  yield takeLatest('my-action', helperWorker1, 1)
  yield takeLatest('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLatest('my-action', helperSaga1)
  // $ExpectError
  yield takeLatest('my-action', helperSaga1, 1)
  yield takeLatest('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLatest(isMyAction, action => action.customField)

  yield takeLatest(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLatest(() => {}, (action: Action) => {})

  yield takeLatest(stringableActionCreator, action => action.customField)

  yield takeLatest(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLatest(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLatest([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLatest(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
```

```
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest(channel)

  // $ExpectError
  yield takeLatest(channel, (action: Action) => {})
  yield takeLatest(channel, (action: ChannelItem) => {})
  yield takeLatest(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLatest(channel, helperWorker1)
  // $ExpectError
  yield takeLatest(channel, helperWorker1, 1)
  yield takeLatest(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga1)
  // $ExpectError
  yield takeLatest(channel, helperSaga1, 1)
  yield takeLatest(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeLatest(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest(eventChannel, (action: ChannelItem) => {})
  yield takeLatest(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading()
  // $ExpectError
  yield takeLeading('my-action')

  yield takeLeading('my-action', (action: Action) => {})
  yield takeLeading('my-action', (action: MyAction) => {})
  yield takeLeading('my-action', function*(action: Action): SagaIterator {})
  yield takeLeading('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker1)
  // $ExpectError
  yield takeLeading('my-action', helperWorker1, 1)
  yield takeLeading('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga1)
  // $ExpectError
  yield takeLeading('my-action', helperSaga1, 1)
  yield takeLeading('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLeading(isMyAction, action => action.customField)

  yield takeLeading(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLeading(() => {}, (action: Action) => {})

  yield takeLeading(stringableActionCreator, action => action.customField)

  yield takeLeading(
    stringableActionCreator,
```

```
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLeading(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLeading([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLeading(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading(channel)

  // $ExpectError
  yield takeLeading(channel, (action: Action) => {})
  yield takeLeading(channel, (action: ChannelItem) => {})
  yield takeLeading(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLeading(channel, helperWorker1)
  // $ExpectError
  yield takeLeading(channel, helperWorker1, 1)
  yield takeLeading(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga1)
  // $ExpectError
  yield takeLeading(channel, helperSaga1, 1)
  yield takeLeading(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeLeading(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading(eventChannel, (action: ChannelItem) => {})
  yield takeLeading(multicastChannel, (action: ChannelItem) => {})
}

function* testThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1)
  // $ExpectError
  yield throttle(1, 'my-action')

  yield throttle(1, 'my-action', (action: Action) => {})
  yield throttle(1, 'my-action', (action: MyAction) => {})
  yield throttle(1, 'my-action', function*(action: Action): SagaIterator {})
  yield throttle(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1)
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1, 1)
  yield throttle(1, 'my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga1)
  // $ExpectError
```

```
  yield throttle(1, 'my-action', helperSaga1, 1)
  yield throttle(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield throttle(1, isMyAction, action => action.customField)

  yield throttle(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield throttle(1, () => {}, (action: Action) => {})

  yield throttle(1, stringableActionCreator, action => action.customField)

  yield throttle(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield throttle(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield throttle(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield throttle(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1, channel)

  // $ExpectError
  yield throttle(1, channel, (action: Action) => {})
  yield throttle(1, channel, (action: ChannelItem) => {})
  yield throttle(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker1)
  // $ExpectError
  yield throttle(1, channel, helperWorker1, 1)
  yield throttle(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga1)
  // $ExpectError
  yield throttle(1, channel, helperSaga1, 1)
  yield throttle(1, channel, helperSaga1, 'a')
```

```
  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, eventChannel, (action: ChannelItem) => {})
  yield throttle(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1)
  // $ExpectError
  yield debounce(1, 'my-action')

  yield debounce(1, 'my-action', (action: Action) => {})
  yield debounce(1, 'my-action', (action: MyAction) => {})
  yield debounce(1, 'my-action', function*(action: Action): SagaIterator {})
  yield debounce(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1)
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1, 1)
  yield debounce(1, 'my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1)
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1, 1)
  yield debounce(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield debounce(1, isMyAction, action => action.customField)

  yield debounce(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield debounce(1, () => {}, (action: Action) => {})

  yield debounce(1, stringableActionCreator, action => action.customField)

  yield debounce(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield debounce(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield debounce(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield debounce(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
```

```
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1, channel)

  // $ExpectError
  yield debounce(1, channel, (action: Action) => {})
  yield debounce(1, channel, (action: ChannelItem) => {})
  yield debounce(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker1)
  // $ExpectError
  yield debounce(1, channel, helperWorker1, 1)
  yield debounce(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield debounce(1, channel, helperSaga1)
  // $ExpectError
  yield debounce(1, channel, helperSaga1, 1)
  yield debounce(1, channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield debounce(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, eventChannel, (action: ChannelItem) => {})
  yield debounce(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDelay(): SagaIterator {
  // $ExpectError
  yield delay()
  yield delay(1)
}

function* testRetry(): SagaIterator {
  // $ExpectError
  yield retry()
  // $ExpectError
  yield retry(1, 0, 1)
  yield retry(1, 0, () => 1)

  yield retry<() => 'foo'>(1, 0, () => 'foo')
  // $ExpectError
  yield retry<() => 'bar'>(1, 0, () => 'foo')

  yield retry(1, 0, a => a + 1, 42)
  // $ExpectError
  yield retry(1, 0, (a: string) => a, 42)
  // $ExpectError
  yield retry<(a: string) => number>(1, 0, a => a, 42)

  yield retry(1, 0, (a: number, b: number, c: string) => a, 1, 2, '3')
}

declare const promise: Promise<any>

function* testAll(): SagaIterator {
  yield all([call(() => {})])

  // $ExpectError
  yield all([1])

  // $ExpectError
  yield all([() => {}])

  // $ExpectError
  yield all([promise])

  // $ExpectError
  yield all([1, () => {}, promise])

  yield all({
    named: call(() => {}),
  })

  // $ExpectError
  yield all({
    named: 1,
  })

  // $ExpectError
  yield all({
```

```
    named: () => {},
  })

  // $ExpectError
  yield all({
    named: promise,
  })

  // $ExpectError
  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testNonStrictAll() {
  yield all([1])

  yield all([() => {}])

  yield all([promise])

  yield all([1, () => {}, promise])

  yield all({
    named: 1,
  })

  yield all({
    named: () => {},
  })

  yield all({
    named: promise,
  })

  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testRace(): SagaIterator {
  yield race({
    call: call(() => {}),
  })

  // $ExpectError
  yield race({
    named: 1,
  })

  // $ExpectError
  yield race({
    named: () => {},
  })

  // $ExpectError
  yield race({
    named: promise,
  })

  // $ExpectError
  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  // $ExpectError
  yield race([...effectArray, promise])
}

function* testNonStrictRace() {
  yield race({
    named: 1,
  })

  yield race({
    named: () => {},
  })

  yield race({
    named: promise,
  })

  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  yield race([...effectArray, promise])
}
```

../redux-saga/packages/core/types/index.d.ts

```
// TypeScript Version: 3.2
import { Saga, Buffer, Channel, END as EndType, Predicate, SagaIterator, Task, NotUndefined } from '@redux-saga/types'
import { ForkEffect } from './effects'

export { Saga, SagaIterator, Buffer, Channel, Task }
```

```
export type Action<T extends string = string> = {
  type: T
}

export interface AnyAction extends Action {
  [extraProps: string]: any
}

export interface UnknownAction extends Action {
  [extraProps: string]: unknown
}

interface Dispatch<A extends Action = UnknownAction> {
  <T extends A>(action: T, ...extraArgs: any[]): T
}

interface MiddlewareAPI<D extends Dispatch = Dispatch, S = any> {
  dispatch: D
  getState(): S
}

export interface Middleware<_DispatchExt = {}, S = any, D extends Dispatch = Dispatch> {
  (api: MiddlewareAPI<D, S>): (next: (action: never) => unknown) => (action: unknown) => unknown
}

/**
 * Used by the middleware to dispatch monitoring events. Actually the middleware
 * dispatches 6 events:
 *
 * - When a root saga is started (via `runSaga` or `sagaMiddleware.run`) the
 *   middleware invokes `sagaMonitor.rootSagaStarted`
 *
 * - When an effect is triggered (via `yield someEffect`) the middleware invokes
 *   `sagaMonitor.effectTriggered`
 *
 * - If the effect is resolved with success the middleware invokes
 *   `sagaMonitor.effectResolved`
 *
 * - If the effect is rejected with an error the middleware invokes
 *   `sagaMonitor.effectRejected`
 *
 * - If the effect is cancelled the middleware invokes
 *   `sagaMonitor.effectCancelled`
 *
 * - Finally, the middleware invokes `sagaMonitor.actionDispatched` when a Redux
 *   action is dispatched.
 */
export interface SagaMonitor {
  /**
   * @param effectId Unique ID assigned to this root saga execution
   * @param saga The generator function that starts to run
   * @param args The arguments passed to the generator function
   */
  rootSagaStarted?(options: { effectId: number; saga: Saga; args: any[] }): void
  /**
   * @param effectId Unique ID assigned to the yielded effect
   * @param parentEffectId ID of the parent Effect. In the case of a `race` or
   *   `parallel` effect, all effects yielded inside will have the direct
   *   race/parallel effect as a parent. In case of a top-level effect, the
   *   parent will be the containing Saga
   * @param label In case of a `race`/`all` effect, all child effects will be
   *   assigned as label the corresponding keys of the object passed to
   *   `race`/`all`
   * @param effect The yielded effect itself
   */
  effectTriggered?(options: { effectId: number; parentEffectId: number; label?: string; effect: any }): void
  /**
   * @param effectId The ID of the yielded effect
   * @param result The result of the successful resolution of the effect. In
   *   case of `fork` or `spawn` effects, the result will be a `Task` object.
   */
  effectResolved?(effectId: number, result: any): void
  /**
   * @param effectId The ID of the yielded effect
   * @param error Error raised with the rejection of the effect
   */
  effectRejected?(effectId: number, error: any): void
  /**
   * @param effectId The ID of the yielded effect
   */
  effectCancelled?(effectId: number): void
  /**
   * @param action The dispatched Redux action. If the action was dispatched by
   * a Saga then the action will have a property `SAGA_ACTION` set to true
   * (`SAGA_ACTION` can be imported from `@redux-saga/symbols`).
   */
  actionDispatched?(action: Action): void
}

/**
 * Creates a Redux middleware and connects the Sagas to the Redux Store
 *
 * #### Example
 *
 * Below we will create a function `configureStore` which will enhance the Store
 * with a new method `runSaga`. Then in our main module, we will use the method
 * to start the root Saga of the application.
 *
 * **configureStore.js**
 *
 *    import createSagaMiddleware from 'redux-saga'
 *    import reducer from './path/to/reducer'
 *
 *    export default function configureStore(initialState) {
 *      // Note: passing middleware as the last argument to createStore requires redux@>=3.1.0
 *      const sagaMiddleware = createSagaMiddleware()
 *      return {
 *        ...createStore(reducer, initialState, applyMiddleware(... other middleware ..., sagaMiddleware)),
 *        runSaga: sagaMiddleware.run
 *      }
 *    }
 *
```

```
 * **main.js**
 *
 *    import configureStore from './configureStore'
 *    import rootSaga from './sagas'
 *    // ... other imports
 *
 *    const store = configureStore()
 *    store.runSaga(rootSaga)
 *
 * @param options A list of options to pass to the middleware
 */
export default function createSagaMiddleware<C extends object>(options?: SagaMiddlewareOptions<C>): SagaMiddleware<C>

export interface SagaMiddlewareOptions<C extends object = {}> {
  /**
   * Initial value of the saga's context.
   */
  context?: C
  /**
   * If a Saga Monitor is provided, the middleware will deliver monitoring
   * events to the monitor.
   */
  sagaMonitor?: SagaMonitor
  /**
   * If provided, the middleware will call it with uncaught errors from Sagas.
   * useful for sending uncaught exceptions to error tracking services.
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * Allows you to intercept any effect, resolve it on your own and pass to the
   * next middleware.
   */
  effectMiddlewares?: EffectMiddleware[]
  /**
   * If provided, the middleware will use this channel instead of the default `stdChannel` for
   * take and put effects.
   */
  channel?: MulticastChannel<Action>
}

export interface SagaMiddleware<C extends object = {}> extends Middleware {
  /**
   * Dynamically run `saga`. Can be used to run Sagas **only after** the
   * `applyMiddleware` phase.
   *
   * The method returns a `Task` descriptor.
   *
   * #### Notes
   *
   * `saga` must be a function which returns a [Generator
   * Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator).
   * The middleware will then iterate over the Generator and execute all yielded
   * Effects.
   *
   * `saga` may also start other sagas using the various Effects provided by the
   * library. The iteration process described below is also applied to all child
   * sagas.
   *
   * In the first iteration, the middleware invokes the `next()` method to
   * retrieve the next Effect. The middleware then executes the yielded Effect
   * as specified by the Effects API below. Meanwhile, the Generator will be
   * suspended until the effect execution terminates. Upon receiving the result
   * of the execution, the middleware calls `next(result)` on the Generator
   * passing it the retrieved result as an argument. This process is repeated
   * until the Generator terminates normally or by throwing some error.
   *
   * If the execution results in an error (as specified by each Effect creator)
   * then the `throw(error)` method of the Generator is called instead. If the
   * Generator function defines a `try/catch` surrounding the current yield
   * instruction, then the `catch` block will be invoked by the underlying
   * Generator runtime. The runtime will also invoke any corresponding finally
   * block.
   *
   * In the case a Saga is cancelled (either manually or using the provided
   * Effects), the middleware will invoke `return()` method of the Generator.
   * This will cause the Generator to skip directly to the finally block.
   *
   * @param saga a Generator function
   * @param args arguments to be provided to `saga`
   */
  run<S extends Saga>(saga: S, ...args: Parameters<S>): Task

  setContext(props: Partial<C>): void
}

export interface EffectMiddleware {
  (next: (effect: any) => void): (effect: any) => void
}

/**
 * Allows starting sagas outside the Redux middleware environment. Useful if you
 * want to connect a Saga to external input/output, other than store actions.
 *
 * `runSaga` returns a Task object. Just like the one returned from a `fork`
 * effect.
 */
export function runSaga<Action, State, S extends Saga>(
  options: RunSagaOptions<Action, State>,
  saga: S,
  ...args: Parameters<S>
): Task

interface ErrorInfo {
  sagaStack: string
}

/**
 * The `{subscribe, dispatch}` is used to fulfill `take` and `put` Effects. This
 * defines the Input/Output interface of the Saga.
 *
 * `subscribe` is used to fulfill `take(PATTERN)` effects. It must call
 * `callback` every time it has an input to dispatch (e.g. on every mouse click
```

```
 * if the Saga is connected to DOM click events). Each time `subscribe` emits an
 * input to its callbacks, if the Saga is blocked on a `take` effect, and if the
 * take pattern matches the currently incoming input, the Saga is resumed with
 * that input.
 *
 * `dispatch` is used to fulfill `put` effects. Each time the Saga emits a
 * `yield put(output)`, `dispatch` is invoked with output.
 */
export interface RunSagaOptions<A, S> {
  /**
   * See docs for `channel`
   */
  channel?: PredicateTakeableChannel<A>
  /**
   * Used to fulfill `put` effects.
   *
   * @param output argument provided by the Saga to the `put` Effect
   */
  dispatch?(output: A): any
  /**
   * Used to fulfill `select` and `getState` effects
   */
  getState?(): S
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  sagaMonitor?: SagaMonitor
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  context?: object
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  effectMiddlewares?: EffectMiddleware[]
}

export const CANCEL: string
export const END: EndType
export type END = EndType

export interface TakeableChannel<T> {
  take(cb: (message: T | END) => void): void
}

export interface PuttableChannel<T> {
  put(message: T | END): void
}

export interface FlushableChannel<T> {
  flush(cb: (items: T[] | END) => void): void
}

/**
 * A factory method that can be used to create Channels. You can optionally pass
 * it a buffer to control how the channel buffers the messages.
 *
 * By default, if no buffer is provided, the channel will queue incoming
 * messages up to 10 until interested takers are registered. The default
 * buffering will deliver message using a FIFO strategy: a new taker will be
 * delivered the oldest message in the buffer.
 */
export function channel<T extends NotUndefined>(buffer?: Buffer<T>): Channel<T>

/**
 * Creates channel that will subscribe to an event source using the `subscribe`
 * method. Incoming events from the event source will be queued in the channel
 * until interested takers are registered.
 *
 * To notify the channel that the event source has terminated, you can notify
 * the provided subscriber with an `END`
 *
 * #### Example
 *
 * In the following example we create an event channel that will subscribe to a
 * `setInterval`
 *
 *    const countdown = (secs) => {
 *      return eventChannel(emitter => {
 *          const iv = setInterval(() => {
 *            console.log('countdown', secs)
 *            secs -= 1
 *            if (secs > 0) {
 *              emitter(secs)
 *            } else {
 *              emitter(END)
 *              clearInterval(iv)
 *              console.log('countdown terminated')
 *            }
 *          }, 1000);
 *          return () => {
 *            clearInterval(iv)
 *            console.log('countdown cancelled')
 *          }
 *        }
 *      )
 *    }
 *
 * @param subscribe used to subscribe to the underlying event source. The
 *   function must return an unsubscribe function to terminate the subscription.
 * @param buffer optional Buffer object to buffer messages on this channel. If
 *   not provided, messages will not be buffered on this channel.
 */
export function eventChannel<T extends NotUndefined>(subscribe: Subscribe<T>, buffer?: Buffer<T>): EventChannel<T>

export type Subscribe<T> = (cb: (input: T | END) => void) => Unsubscribe
export type Unsubscribe = () => void
```

```
export interface EventChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void): void
  flush(cb: (items: T[] | END) => void): void
  close(): void
}

export interface PredicateTakeableChannel<T> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
}

export interface MulticastChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
  put(message: T | END): void
  close(): void
}

export function multicastChannel<T extends NotUndefined>(): MulticastChannel<T>
export function stdChannel<T extends NotUndefined>(): MulticastChannel<T>

export function detach(forkEffect: ForkEffect): ForkEffect

/**
 * Provides some common buffers
 */
export const buffers: {
  /**
   * No buffering, new messages will be lost if there are no pending takers
   */
  none<T>(): Buffer<T>
  /**
   * New messages will be buffered up to `limit`. Overflow will raise an Error.
   * Omitting a `limit` value will result in a limit of 10.
   */
  fixed<T>(limit?: number): Buffer<T>
  /**
   * Like `fixed` but Overflow will cause the buffer to expand dynamically.
   */
  expanding<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will silently drop the messages.
   */
  dropping<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will insert the new message at the end and
   * drop the oldest message in the buffer.
   */
  sliding<T>(limit?: number): Buffer<T>
}
```

../redux-saga/packages/core/types/middleware.test.ts

```
import createSagaMiddleware, { SagaIterator } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'
import { applyMiddleware } from 'redux'

function testApplyMiddleware() {
  const middleware = createSagaMiddleware()

  const enhancer = applyMiddleware(middleware)
}

declare const effect: StrictEffect
declare const promise: Promise<any>

function testRun() {
  const middleware = createSagaMiddleware()

  middleware.run(function* saga(): SagaIterator {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // middleware.run(function* saga(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a'): SagaIterator {}, 1)

  middleware.run(function* saga(a: 'a'): SagaIterator {}, 'a')

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

  // test with any iterator i.e. when generator doesn't always yield Effects.
  middleware.run(function* saga() {
    yield promise
  })
}

function testOptions() {
  const emptyOptions = createSagaMiddleware({})

  const withOptions = createSagaMiddleware({
    onError(error) {
      console.error(error)
    },

    sagaMonitor: {
```

```
    effectTriggered() {},
  },

  effectMiddlewares: [
    next => effect => {
      setTimeout(() => {
        next(effect)
      }, 10)
    },
    next => effect => {
      setTimeout(() => {
        next(effect)
      }, 10)
    },
  ],
})

const withMonitor = createSagaMiddleware({
  sagaMonitor: {
    effectTriggered() {},
    effectResolved() {},
    effectRejected() {},
    effectCancelled() {},
    actionDispatched() {},
  },
})
}

function testContext() {
  interface Context {
    a: string
    b: number
  }

  // $ExpectError
  createSagaMiddleware<Context>({ context: { c: 42 } })

  // $ExpectError
  createSagaMiddleware({ context: 42 })

  const middleware = createSagaMiddleware<Context>({
    context: { a: '', b: 42 },
  })

  // $ExpectError
  middleware.setContext({ c: 42 })

  middleware.setContext({ b: 42 })

  const task = middleware.run(function*() {
    yield effect
  })
  task.setContext({ b: 42 })

  task.setContext<Context>({ a: '' })
  // $ExpectError
  task.setContext<Context>({ c: '' })
}
```

../redux-saga/packages/core/types/runSaga.test.ts

```
import { SagaIterator, Task, runSaga, END, MulticastChannel } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'

declare const stdChannel: MulticastChannel<any>
declare const promise: Promise<any>
declare const effect: StrictEffect
declare const iterator: Iterator<any>

function testRunSaga() {
  const task0: Task = runSaga<{ foo: string }, { baz: boolean }, () => SagaIterator>(
    {
      context: { a: 42 },

      channel: stdChannel,

      effectMiddlewares: [
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
      ],

      getState() {
        return { baz: true }
      },

      dispatch(input) {
        input.foo
        // $ExpectError
        input.bar
      },

      sagaMonitor: {
        effectTriggered() {},
        effectResolved() {},
        effectRejected() {},
        effectCancelled() {},
        actionDispatched() {},
      },

      onError(error) {
        console.error(error)
```

```
    },
  },
  function* saga(): SagaIterator {
    yield effect
  },
)

// $ExpectError
runSaga()

// $ExpectError
runSaga({})

// $ExpectError
runSaga({}, iterator)

runSaga({}, function* saga() {
  yield effect
})

// TODO: https://github.com/Microsoft/TypeScript/issues/28803
{
  // // $ExpectError
  // runSaga({}, function* saga(a: 'a'): SagaIterator {})
}

// $ExpectError
runSaga({}, function* saga(a: 'a'): SagaIterator {}, 1)

runSaga({}, function* saga(a: 'a'): SagaIterator {}, 'a')

// TODO: https://github.com/Microsoft/TypeScript/issues/28803
{
  // // $ExpectError
  // runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
}

// $ExpectError
runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

// $ExpectError
runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

// test with any iterator i.e. when generator doesn't always yield Effects.
runSaga({}, function* saga() {
  yield promise
})

// $ExpectError
runSaga({ context: 42 }, function* saga(): SagaIterator {})
}
```

../redux-saga/packages/core/types/ts3.6/channels.test.ts

```
import {
  buffers, Buffer, channel, Channel, EventChannel, MulticastChannel, END,
  eventChannel, multicastChannel, stdChannel,
} from "redux-saga";

function testBuffers() {
  const b1: Buffer<{foo: string}> = buffers.none<{foo: string}>();

  const b2: Buffer<{foo: string}> = buffers.dropping<{foo: string}>();
  const b3: Buffer<{foo: string}> = buffers.dropping<{foo: string}>(42);

  const b4: Buffer<{foo: string}> = buffers.expanding<{foo: string}>();
  const b5: Buffer<{foo: string}> = buffers.expanding<{foo: string}>(42);

  const b6: Buffer<{foo: string}> = buffers.fixed<{foo: string}>();
  const b7: Buffer<{foo: string}> = buffers.fixed<{foo: string}>(42);

  const b8: Buffer<{foo: string}> = buffers.sliding<{foo: string}>();
  const b9: Buffer<{foo: string}> = buffers.sliding<{foo: string}>(42);

  const buffer = buffers.none<{foo: string}>();

  // $ExpectError
  buffer.put({bar: 'bar'});
  buffer.put({foo: 'foo'});

  const isEmpty: boolean = buffer.isEmpty();

  const item = buffer.take();

  // $ExpectError
  item.foo;  // item may be undefined

  const foo: string = item!.foo;

  if (buffer.flush)
    buffer.flush();
}

function testChannel() {
  const c1: Channel<{foo: string}> = channel<{foo: string}>();
  const c2: Channel<{foo: string}> = channel(buffers.none<{foo: string}>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);
```

```
  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: Array<{bar: number}> | END) => {});
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();

  // Testing that we can't define channels that pass void or undefined
  // $ExpectError
  const voidChannel: Channel<void> = channel();
  // $ExpectError
  const voidChannel2 = channel<void>();
  // $ExpectError
  const undefinedChannel = channel<undefined>();
  // $ExpectError
  channel().put();
  // $ExpectError
  channel().put(undefined);

  // Testing that we can pass primitives into channels
  channel().put(null);
  channel().put(42);
  channel().put('test');
  channel().put(true);
}

function testEventChannel(secs: number) {
  const subscribe = (emitter: (input: number | END) => void) => {
    const iv = setInterval(() => {
      secs -= 1
      if (secs > 0) {
        emitter(secs)
      } else {
        emitter(END)
        clearInterval(iv)
      }
    }, 1000);
    return () => {
      clearInterval(iv)
    }
  };
  const c1: EventChannel<number> = eventChannel<number>(subscribe);

  const c2: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<string>()); // $ExpectError

  const c3: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<number>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: string | END) => {});
  c1.take((message: number | END) => {});

  // $ExpectError
  c1.put(1);

  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: string[] | END) => {});
  c1.flush((messages: number[] | END) => {});

  c1.close();

  // $ExpectError
  const c4: EventChannel<void> = eventChannel(() => () => {})

  // $ExpectError
  const c5 = eventChannel<void>(emit => {
    emit()
    return () => {}
  })

  const c6 = eventChannel(emit => {
    // $ExpectError
    emit()
    return () => {}
  })
}

function testMulticastChannel() {
  const c1: MulticastChannel<{foo: string}> = multicastChannel<{foo: string}>();
  const c2: MulticastChannel<{foo: string}> = stdChannel<{foo: string}>();
  // $ExpectError
  const c3: MulticastChannel<void> = stdChannel()
  // $ExpectError
  const c4 = multicastChannel<void>()
  // $ExpectError
  const c5 = stdChannel<void>()

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);

  // $ExpectError
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();
}
```

../redux-saga/packages/core/types/ts3.6/effects.d.ts

```
import { Last, Reverse } from 'typescript-tuple'

import {
  ActionPattern,
  Effect,
  Buffer,
  CombinatorEffect,
  CombinatorEffectDescriptor,
  SimpleEffect,
  END,
  Pattern,
  Predicate,
  Task,
  StrictEffect,
  ActionMatchingPattern,
  SagaIterator,
} from '@redux-saga/types'

import { FlushableChannel, PuttableChannel, TakeableChannel, Action, AnyAction } from './index'

export { ActionPattern, Effect, Pattern, SimpleEffect, StrictEffect }

export const effectTypes: {
  TAKE: 'TAKE'
  PUT: 'PUT'
  ALL: 'ALL'
  RACE: 'RACE'
  CALL: 'CALL'
  CPS: 'CPS'
  FORK: 'FORK'
  JOIN: 'JOIN'
  CANCEL: 'CANCEL'
  SELECT: 'SELECT'
  ACTION_CHANNEL: 'ACTION_CHANNEL'
  CANCELLED: 'CANCELLED'
  FLUSH: 'FLUSH'
  GET_CONTEXT: 'GET_CONTEXT'
  SET_CONTEXT: 'SET_CONTEXT'
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
 * specified action on the Store. The Generator is suspended until an action
 * that matches `pattern` is dispatched.
 *
 * The result of `yield take(pattern)` is an action object being dispatched.
 *
 * `pattern` is interpreted using the following rules:
 *
 * - If `take` is called with no arguments or `'*'` all dispatched actions are
 *   matched (e.g. `take()` will match all actions)
 *
 * - If it is a function, the action is matched if `pattern(action)` is true
 *   (e.g. `take(action => action.entities)` will match all actions having a
 *   (truthy) `entities` field.)
 * > Note: if the pattern function has `toString` defined on it, `action.type`
 * > will be tested against `pattern.toString()` instead. This is useful if
 * > you're using an action creator library like redux-act or redux-actions.
 *
 * - If it is a String, the action is matched if `action.type === pattern` (e.g.
 *   `take(INCREMENT_ASYNC)`
 *
 * - If it is an array, each item in the array is matched with aforementioned
 *   rules, so the mixed array of strings and function predicates is supported.
 *   The most common use case is an array of strings though, so that
 *   `action.type` is matched against all items in the array (e.g.
 *   `take([INCREMENT, DECREMENT])` and that would match either actions of type
 *   `INCREMENT` or `DECREMENT`).
 *
 * The middleware provides a special action `END`. If you dispatch the END
 * action, then all Sagas blocked on a take Effect will be terminated regardless
 * of the specified pattern. If the terminated Saga has still some forked tasks
 * which are still running, it will wait for all the child tasks to terminate
 * before terminating the Task.
 */
export function take(pattern?: ActionPattern): TakeEffect
export function take<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

/**
 * Same as `take(pattern)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 *
 * #### Notes
 *
 * `takeMaybe` got its name from the FP analogy - it's like instead of having a
 * return type of `ACTION` (with automatic handling) we can have a type of
 * `Maybe(ACTION)` so we can handle both cases:
 *
 * - case when there is a `Just(ACTION)` (we have an action)
 * - the case of `NOTHING` (channel was closed*). i.e. we need some way to map
 *   over `END`
 *
 * internally all `dispatch`ed actions are going through the `stdChannel` which
 * is getting closed when `dispatch(END)` happens
 */
export function takeMaybe(pattern?: ActionPattern): TakeEffect
export function takeMaybe<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

export type TakeEffect = SimpleEffect<'TAKE', TakeEffectDescriptor>

export interface TakeEffectDescriptor {
  pattern: ActionPattern
  maybe?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
```

```
 * specified message from the provided Channel. If the channel is already
 * closed, then the Generator will immediately terminate following the same
 * process described above for `take(pattern)`.
 */
export function take<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

/**
 * Same as `take(channel)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 */
export function takeMaybe<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

export type ChannelTakeEffect<T> = SimpleEffect<'TAKE', ChannelTakeEffectDescriptor<T>>

export interface ChannelTakeEffectDescriptor<T> {
  channel: TakeableChannel<T>
  pattern?: Pattern<T>
  maybe?: boolean
}

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeEvery` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action:
 *
 *    import { takeEvery } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchFetchUser() {
 *      yield takeEvery('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeEvery` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeEvery = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * `takeEvery` allows concurrent actions to be handled. In the example above,
 * when a `USER_REQUESTED` action is dispatched, a new `fetchUser` task is
 * started even if a previous `fetchUser` is still pending (for example, the
 * user clicks on a `Load User` button 2 consecutive times at a rapid rate, the
 * 2nd click will dispatch a `USER_REQUESTED` action while the `fetchUser` fired
 * on the first one hasn't yet terminated)
 *
 * `takeEvery` doesn't handle out of order responses from tasks. There is no
 * guarantee that the tasks will terminate in the same order they were started.
 * To handle out of order responses, you may consider `takeLatest` below.
 *
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeEvery` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeEvery<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function takeEvery<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeEvery<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeEvery<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeEvery(pattern, saga, ...args)`.
 */
export function takeEvery<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeEvery<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. And automatically cancels any previous `saga` task started
 * previously if it's still running.
 *
 * Each time an action is dispatched to the store. And if this action matches
 * `pattern`, `takeLatest` starts a new `saga` task in the background. If a
 * `saga` task was started previously (on the last action dispatched before the
 * actual action), and if this task is still running, the task will be
 * cancelled.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLatest` to start a new `fetchUser` task on each dispatched
```

```
 * `USER_REQUESTED` action. Since `takeLatest` cancels any pending task started
 * previously, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only conclude with the latest action
 *
 *    import { takeLatest } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLatest('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLatest` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeLatest = (patternOrChannel, saga, ...args) => fork(function*() {
 *      let lastTask
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        if (lastTask) {
 *          yield cancel(lastTask) // cancel is no-op if the task has already terminated
 *        }
 *        lastTask = yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLatest` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeLatest<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function takeLatest<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeLatest<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeLatest<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLatest(pattern, saga, ...args)`.
 */
export function takeLatest<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeLatest<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. After spawning a task once, it blocks until spawned saga completes
 * and then starts to listen for a `pattern` again.
 *
 * In short, `takeLeading` is listening for the actions when it doesn't run a
 * saga.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLeading` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action. Since `takeLeading` ignores any new coming task
 * after it's started, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only keep on running with the leading
 * action
 *
 *    import { takeLeading } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLeading('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLeading` is a high-level API built using `take` and `call`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeLeading = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel);
 *        yield call(saga, ...args.concat(action));
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLeading` will
 *   add the incoming action to the argument list (i.e. the action will be the
 *   last argument provided to `saga`)
 */
export function takeLeading<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
```

```
export function takeLeading<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeLeading<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeLeading<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLeading(pattern, saga, ...args)`.
 */
export function takeLeading<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeLeading<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

export type HelperWorkerParameters<T, Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends T
  ? AllButLast<Parameters<Fn>>
  : Parameters<Fn>

interface ThunkDispatch<State, ExtraThunkArg, BasicAction extends Action> {
  <ReturnType>(thunkAction: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>): ReturnType
  <Action extends BasicAction>(action: Action): Action
  <ReturnType, Action extends BasicAction>(
    action: Action | ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
  ): Action | ReturnType
}

export type ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction extends Action> = (
  dispatch: ThunkDispatch<State, ExtraThunkArg, BasicAction>,
  getState: () => State,
  extraArgument: ExtraThunkArg,
) => ReturnType

/**
 * Creates an Effect description that instructs the middleware to dispatch an
 * action to the Store. This effect is non-blocking, any errors that are
 * thrown downstream (e.g. in a reducer) will bubble back into the saga.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<A extends Action>(action: A): PutEffect<A>
export function put<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

/**
 * Just like `put` but the effect is blocking (if promise is returned from
 * `dispatch` it will wait for its resolution) and will bubble up errors from
 * downstream.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function putResolve<A extends Action>(action: A): PutEffect<A>
export function putResolve<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

export type PutEffect<A extends Action = AnyAction> = SimpleEffect<'PUT', PutEffectDescriptor<A>>

export interface PutEffectDescriptor<A extends Action> {
  action: A
  channel: null
  resolve?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to put an action
 * into the provided channel.
 *
 * This effect is blocking if the put is *not* buffered but immediately consumed
 * by takers. If an error is thrown in any of these takers it will bubble back
 * into the saga.
 *
 * @param channel a `Channel` Object.
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<T>(channel: PuttableChannel<T>, action: T | END): ChannelPutEffect<T>

export type ChannelPutEffect<T> = SimpleEffect<'PUT', ChannelPutEffectDescriptor<T>>

export interface ChannelPutEffectDescriptor<T> {
  action: T
  channel: PuttableChannel<T>
}

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments.
 *
 * #### Notes
 *
 * `fn` can be either a *normal* or a Generator function.
 *
 * The middleware invokes the function and examines its result.
 *
 * If the result is an Iterator object, the middleware will run that Generator
 * function, just like it did with the startup Generators (passed to the
 * middleware on startup). The parent Generator will be suspended until the
 * child Generator terminates normally, in which case the parent Generator is
 * resumed with the value returned by the child Generator. Or until the child
 * aborts with some error, in which case an error will be thrown inside the
 * parent Generator.
 *
 * If `fn` is a normal function and returns a Promise, the middleware will
```

```
 * suspend the Generator until the Promise is settled. After the promise is
 * resolved the Generator is resumed with the resolved value, or if the Promise
 * is rejected an error is thrown inside the Generator.
 *
 * If the result is not an Iterator object nor a Promise, the middleware will
 * immediately return that value back to the saga, so that it can resume its
 * execution synchronously.
 *
 * When an error is thrown inside the Generator, if it has a `try/catch` block
 * surrounding the current `yield` instruction, the control will be passed to
 * the `catch` block. Otherwise, the Generator aborts with the raised error, and
 * if this Generator was called by another Generator, the error will propagate
 * to the calling Generator.
 *
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
export function call<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>
/**
 * Same as `call([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield call([localStorage, 'getItem'], 'redux-saga')`
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): CallEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): CallEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `call(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>

export type CallEffect<RT = any> = SimpleEffect<'CALL', CallEffectDescriptor<RT>>

export interface CallEffectDescriptor<RT> {
  context: any
  fn: (...args: any[]) => SagaIterator<RT> | Promise<RT> | RT
  args: any[]
}

export type SagaReturnType<S extends Function> = S extends (...args: any[]) => SagaIterator<infer RT>
  ? RT
  : S extends (...args: any[]) => Promise<infer RT>
  ? RT
  : S extends (...args: any[]) => infer RT
  ? RT
  : never

/**
 * Alias for `call([context, fn], ...args)`.
 */
export function apply<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctx: Ctx,
  fnName: Name,
  args: Parameters<Ctx[Name]>,
): CallEffect<SagaReturnType<Ctx[Name]>>
export function apply<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctx: Ctx,
  fn: Fn,
  args: Parameters<Fn>,
): CallEffect<SagaReturnType<Fn>>

type Cast<A, B> = A extends B ? A : B
type AnyFunction = (...args: any[]) => any

type RequireCpsCallback<Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends CpsCallback<any> ? Fn : never
type RequireCpsNamedCallback<Ctx, Name extends keyof Ctx> = Last<
  Parameters<Cast<Ctx[Name], AnyFunction>>
> extends CpsCallback<any>
  ? Name
  : never

/**
 * Creates an Effect description that instructs the middleware to invoke `fn` as
 * a Node style function.
 *
 * @param fn a Node style function. i.e. a function which accepts in addition to
 *   its arguments, an additional callback to be invoked by `fn` when it
 *   terminates. The callback accepts two parameters, where the first parameter
 *   is used to report errors while the second is used to report successful
 *   results
 * @param args an array to be passed as arguments for `fn`
 */
export function cps<Fn extends (cb: CpsCallback<any>) => any>(fn: Fn): CpsEffect<ReturnType<Fn>>
```

```
export function cps<Fn extends (...args: any[]) => any>(
  fn: RequireCpsCallback<Fn>,
  ...args: AllButLast<Parameters<Fn>>
): CpsEffect<ReturnType<Fn>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield cps([localStorage, 'getItem'], 'redux-saga')`
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: [Ctx, RequireCpsNamedCallback<Ctx, Name>],
  ...args: AllButLast<Parameters<Ctx[Name]>>
): CpsEffect<ReturnType<Ctx[Name]>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: RequireCpsNamedCallback<Ctx, Name> },
  ...args: AllButLast<Parameters<Ctx[Name]>>
): CpsEffect<ReturnType<Ctx[Name]>>
/**
 * Same as `cps(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: [Ctx, RequireCpsCallback<Fn>],
  ...args: AllButLast<Parameters<Fn>>
): CpsEffect<ReturnType<Fn>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: { context: Ctx; fn: RequireCpsCallback<Fn> },
  ...args: AllButLast<Parameters<Fn>>
): CpsEffect<ReturnType<Fn>>

export type CpsFunctionParameters<Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends CpsCallback<any>
  ? AllButLast<Parameters<Fn>>
  : never

export interface CpsCallback<R> {
  (error: any, result: R): void
  cancel?(): void
}

export type CpsEffect<RT> = SimpleEffect<'CPS', CallEffectDescriptor<RT>>

/**
 * Creates an Effect description that instructs the middleware to perform a
 * *non-blocking call* on `fn`
 *
 * returns a `Task` object.
 *
 * #### Note
 *
 * `fork`, like `call`, can be used to invoke both normal and Generator
 * functions. But, the calls are non-blocking, the middleware doesn't suspend
 * the Generator while waiting for the result of `fn`. Instead as soon as `fn`
 * is invoked, the Generator resumes immediately.
 *
 * `fork`, alongside `race`, is a central Effect for managing concurrency
 * between Sagas.
 *
 * The result of `yield fork(fn ...args)` is a `Task` object.  An object
 * with some useful methods and properties.
 *
 * All forked tasks are *attached* to their parents. When the parent terminates
 * the execution of its own body of instructions, it will wait for all forked
 * tasks to terminate before returning.
 *
 * Errors from child tasks automatically bubble up to their parents. If any
 * forked task raises an uncaught error, then the parent task will abort with
 * the child Error, and the whole Parent's execution tree (i.e. forked tasks +
 * the *main task* represented by the parent's body if it's still running) will
 * be cancelled.
 *
 * Cancellation of a forked Task will automatically cancel all forked tasks that
 * are still executing. It'll also cancel the current Effect where the cancelled
 * task was blocked (if any).
 *
 * If a forked task fails *synchronously* (ie: fails immediately after its
 * execution before performing any async operation), then no Task is returned,
 * instead the parent will be aborted as soon as possible (since both parent and
 * child execute in parallel, the parent will abort as soon as it takes notice
 * of the child failure).
 *
 * To create *detached* forks, use `spawn` instead.
 *
 * @param fn A Generator function, or normal function which returns a Promise as result
 * @param args An array of values to be passed as arguments to `fn`
 */
export function fork<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield fork([localStorage, 'getItem'], 'redux-saga')`
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
```

```
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `fork(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>

export type ForkEffect<RT = any> = SimpleEffect<'FORK', ForkEffectDescriptor<RT>>

export interface ForkEffectDescriptor<RT> extends CallEffectDescriptor<RT> {
  detached?: boolean
}

/**
 * Same as `fork(fn, ...args)` but creates a *detached* task. A detached task
 * remains independent from its parent and acts like a top-level task. The
 * parent will not wait for detached tasks to terminate before returning and all
 * events which may affect the parent or the detached task are completely
 * independents (error, cancellation).
 */
export function spawn<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield spawn([localStorage, 'getItem'], 'redux-saga')`
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `spawn(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>

/**
 * Creates an Effect description that instructs the middleware to wait for the
 * result of a previously forked task.
 *
 * #### Notes
 *
 * `join` will resolve to the same outcome of the joined task (success or
 * error). If the joined task is cancelled, the cancellation will also propagate
 * to the Saga executing the join effect. Similarly, any potential callers of
 * those joiners will be cancelled as well.
 *
 * @param task A `Task` object returned by a previous `fork`
 */
export function join(task: Task): JoinEffect
/**
 * Creates an Effect description that instructs the middleware to wait for the
 * results of previously forked tasks.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function join(tasks: Task[]): JoinEffect

export type JoinEffect = SimpleEffect<'JOIN', JoinEffectDescriptor>

export type JoinEffectDescriptor = Task | Task[]

/**
 * Creates an Effect description that instructs the middleware to cancel a
 * previously forked task.
 *
```

```
 * #### Notes
 *
 * To cancel a running task, the middleware will invoke `return` on the
 * underlying Generator object. This will cancel the current Effect in the task
 * and jump to the finally block (if defined).
 *
 * Inside the finally block, you can execute any cleanup logic or dispatch some
 * action to keep the store in a consistent state (e.g. reset the state of a
 * spinner to false when an ajax request is cancelled). You can check inside the
 * finally block if a Saga was cancelled by issuing a `yield cancelled()`.
 *
 * Cancellation propagates downward to child sagas. When cancelling a task, the
 * middleware will also cancel the current Effect (where the task is currently
 * blocked). If the current Effect is a call to another Saga, it will be also
 * cancelled. When cancelling a Saga, all *attached forks* (sagas forked using
 * `yield fork()`) will be cancelled. This means that cancellation effectively
 * affects the whole execution tree that belongs to the cancelled task.
 *
 * `cancel` is a non-blocking Effect. i.e. the Saga executing it will resume
 * immediately after performing the cancellation.
 *
 * For functions which return Promise results, you can plug your own
 * cancellation logic by attaching a `[CANCEL]` to the promise.
 *
 * The following example shows how to attach cancellation logic to a Promise
 * result:
 *
 *    import { CANCEL } from 'redux-saga'
 *    import { fork, cancel } from 'redux-saga/effects'
 *
 *    function myApi() {
 *      const promise = myXhr(...)
 *
 *      promise[CANCEL] = () => myXhr.abort()
 *      return promise
 *    }
 *
 *    function* mySaga() {
 *
 *      const task = yield fork(myApi)
 *
 *      // ... later
 *      // will call promise[CANCEL] on the result of myApi
 *      yield cancel(task)
 *    }
 *
 * redux-saga will automatically cancel jqXHR objects using their `abort` method.
 *
 * @param task A `Task` object returned by a previous `fork`
 */
export function cancel(task: Task): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel
 * previously forked tasks.
 *
 * #### Notes
 *
 * It wraps the array of tasks in cancel effects, roughly becoming the
 * equivalent of `yield tasks.map(t => cancel(t))`.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function cancel(tasks: Task[]): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel a task
 * in which it has been yielded (self-cancellation). It allows to reuse
 * destructor-like logic inside a `finally` blocks for both outer
 * (`cancel(task)`) and self (`cancel()`) cancellations.
 *
 * #### Example
 *
 *    function* deleteRecord({ payload }) {
 *      try {
 *        const { confirm, deny } = yield call(prompt);
 *        if (confirm) {
 *          yield put(actions.deleteRecord.confirmed())
 *        }
 *        if (deny) {
 *          yield cancel()
 *        }
 *      } catch(e) {
 *        // handle failure
 *      } finally {
 *        if (yield cancelled()) {
 *          // shared cancellation logic
 *          yield put(actions.deleteRecord.cancel(payload))
 *        }
 *      }
 *    }
 */
export function cancel(): CancelEffect

export type CancelEffect = SimpleEffect<'CANCEL', CancelEffectDescriptor>

export type CancelEffectDescriptor = Task | Task[] | SELF_CANCELLATION
type SELF_CANCELLATION = '@@redux-saga/SELF_CANCELLATION'

/**
 * Creates an effect that instructs the middleware to invoke the provided
 * selector on the current Store's state (i.e. returns the result of
 * `selector(getState(), ...args)`).
 *
 * If `select` is called without argument (i.e. `yield select()`) then the
 * effect is resolved with the entire state (the same result of a `getState()`
 * call).
 *
 * > It's important to note that when an action is dispatched to the store, the
 * middleware first forwards the action to the reducers and then notifies the
 * Sagas. This means that when you query the Store's State, you get the State
 * **after** the action has been applied. However, this behavior is only
 * guaranteed if all subsequent middlewares call `next(action)` synchronously.
 * If any subsequent middleware calls `next(action)` asynchronously (which is
```

```
 * unusual but possible), then the sagas will get the state from **before** the
 * action is applied.  Therefore it is recommended to review the source of each
 * subsequent middleware to ensure it calls `next(action)` synchronously, or
 * else ensure that redux-saga is the last middleware in the call chain.
 *
 * #### Notes
 *
 * Preferably, a Saga should be autonomous and should not depend on the Store's
 * state. This makes it easy to modify the state implementation without
 * affecting the Saga code. A saga should preferably depend only on its own
 * internal control state when possible. But sometimes, one could find it more
 * convenient for a Saga to query the state instead of maintaining the needed
 * data by itself (for example, when a Saga duplicates the logic of invoking
 * some reducer to compute a state that was already computed by the Store).
 *
 * For example, suppose we have this state shape in our application:
 *
 *    state = {
 *      cart: {...}
 *    }
 *
 * We can create a *selector*, i.e. a function which knows how to extract the
 * `cart` data from the State:
 *
 * `./selectors`
 *
 *    export const getCart = state => state.cart
 *
 *
 * Then we can use that selector from inside a Saga using the `select` Effect:
 *
 * `./sagas.js`
 *
 *    import { take, fork, select } from 'redux-saga/effects'
 *    import { getCart } from './selectors'
 *
 *    function* checkout() {
 *      // query the state using the exported selector
 *      const cart = yield select(getCart)
 *
 *      // ... call some API endpoint then dispatch a success/error action
 *    }
 *
 *    export default function* rootSaga() {
 *      while (true) {
 *        yield take('CHECKOUT_REQUEST')
 *        yield fork(checkout)
 *      }
 *    }
 *
 * `checkout` can get the needed information directly by using
 * `select(getCart)`. The Saga is coupled only with the `getCart` selector. If
 * we have many Sagas (or React Components) that needs to access the `cart`
 * slice, they will all be coupled to the same function `getCart`. And if we now
 * change the state shape, we need only to update `getCart`.
 *
 * @param selector a function `(state, ...args) => args`. It takes the current
 *   state and optionally some arguments and returns a slice of the current
 *   Store's state
 * @param args optional arguments to be passed to the selector in addition of
 *   `getState`.
 */
export function select(): SelectEffect
export function select<Fn extends (state: any, ...args: any[]) => any>(
  selector: Fn,
  ...args: Tail<Parameters<Fn>>
): SelectEffect

export type SelectEffect = SimpleEffect<'SELECT', SelectEffectDescriptor>

export interface SelectEffectDescriptor {
  selector(state: any, ...args: any[]): any
  args: any[]
}

/**
 * Creates an effect that instructs the middleware to queue the actions matching
 * `pattern` using an event channel. Optionally, you can provide a buffer to
 * control buffering of the queued actions.
 *
 * #### Example
 *
 * The following code creates a channel to buffer all `USER_REQUEST` actions.
 * Note that even the Saga may be blocked on the `call` effect. All actions that
 * come while it's blocked are automatically buffered. This causes the Saga to
 * execute the API calls one at a time
 *
 *    import { actionChannel, call } from 'redux-saga/effects'
 *    import api from '...'
 *
 *    function* takeOneAtMost() {
 *      const chan = yield actionChannel('USER_REQUEST')
 *      while (true) {
 *        const {payload} = yield take(chan)
 *        yield call(api.getUser, payload)
 *      }
 *    }
 *
 * @param pattern see API for `take(pattern)`
 * @param buffer a `Buffer` object
 */
export function actionChannel(pattern: ActionPattern, buffer?: Buffer<Action>): ActionChannelEffect

export type ActionChannelEffect = SimpleEffect<'ACTION_CHANNEL', ActionChannelEffectDescriptor>

export interface ActionChannelEffectDescriptor {
  pattern: ActionPattern
  buffer?: Buffer<Action>
}

/**
 * Creates an effect that instructs the middleware to flush all buffered items
```

```
 * from the channel. Flushed items are returned back to the saga, so they can be
 * utilized if needed.
 *
 * #### Example
 *
 *    function* saga() {
 *      const chan = yield actionChannel('ACTION')
 *
 *      try {
 *        while (true) {
 *          const action = yield take(chan)
 *          // ...
 *        }
 *      } finally {
 *        const actions = yield flush(chan)
 *        // ...
 *      }
 *    }
 *
 * @param channel a `Channel` Object.
 */
export function flush<T>(channel: FlushableChannel<T>): FlushEffect<T>

export type FlushEffect<T> = SimpleEffect<'FLUSH', FlushEffectDescriptor<T>>

export type FlushEffectDescriptor<T> = FlushableChannel<T>

/**
 * Creates an effect that instructs the middleware to return whether this
 * generator has been cancelled. Typically you use this Effect in a finally
 * block to run Cancellation specific code
 *
 * #### Example
 *
 *    function* saga() {
 *      try {
 *        // ...
 *      } finally {
 *        if (yield cancelled()) {
 *          // logic that should execute only on Cancellation
 *        }
 *        // logic that should execute in all situations (e.g. closing a channel)
 *      }
 *    }
 */
export function cancelled(): CancelledEffect

export type CancelledEffect = SimpleEffect<'CANCELLED', CancelledEffectDescriptor>

export type CancelledEffectDescriptor = {}

/**
 * Creates an effect that instructs the middleware to update its own context.
 * This effect extends saga's context instead of replacing it.
 */
export function setContext<C extends object>(props: C): SetContextEffect<C>

export type SetContextEffect<C extends object> = SimpleEffect<'SET_CONTEXT', SetContextEffectDescriptor<C>>

export type SetContextEffectDescriptor<C extends object> = C

/**
 * Creates an effect that instructs the middleware to return a specific property
 * of saga's context.
 */
export function getContext(prop: string): GetContextEffect

export type GetContextEffect = SimpleEffect<'GET_CONTEXT', GetContextEffectDescriptor>

export type GetContextEffectDescriptor = string

/**
 * Returns an effect descriptor to block execution for `ms` milliseconds and return `val` value.
 */
export function delay<T = true>(ms: number, val?: T): CallEffect<T>

/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * After spawning a task it's still accepting incoming actions into the
 * underlying `buffer`, keeping at most 1 (the most recent one), but in the same
 * time holding up with spawning new task for `ms` milliseconds (hence its name -
 * `throttle`). Purpose of this is to ignore incoming actions for a given
 * period of time while processing a task.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `throttle` to start a new `fetchAutocomplete` task on dispatched
 * `FETCH_AUTOCOMPLETE` action. However since `throttle` ignores consecutive
 * `FETCH_AUTOCOMPLETE` for some time, we ensure that user won't flood our
 * server with requests.
 *
 *    import { call, put, throttle } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* throttleAutocomplete() {
 *      yield throttle(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `throttle` is a high-level API built using `take`, `fork` and
 * `actionChannel`. Here is how the helper could be implemented using the
 * low-level Effects
 *
 *    const throttle = (ms, pattern, task, ...args) => fork(function*() {
 *      const throttleChannel = yield actionChannel(pattern, buffers.sliding(1))
 *
```

```
 *      while (true) {
 *        const action = yield take(throttleChannel)
 *        yield fork(task, ...args, action)
 *        yield delay(ms)
 *      }
 *    })
 *
 * @param ms length of a time window in milliseconds during which actions will
 *   be ignored after the action starts processing
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `throttle` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function throttle<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function throttle<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function throttle<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect<never>
export function throttle<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `throttle(ms, pattern, saga, ...args)`.
 */
export function throttle<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function throttle<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * Saga will be called after it stops taking `pattern` actions for `ms`
 * milliseconds. Purpose of this is to prevent calling saga until the actions
 * are settled off.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `debounce` to delay calling `fetchAutocomplete` saga until we stop receive
 * any `FETCH_AUTOCOMPLETE` events for at least `1000` ms.
 *
 *    import { call, put, debounce } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* debounceAutocomplete() {
 *      yield debounce(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `debounce` is a high-level API built using `take`, `delay` and `fork`. Here
 * is how the helper could be implemented using the low-level Effects
 *
 *    const debounce = (ms, pattern, task, ...args) => fork(function*() {
 *      while (true) {
 *        let action = yield take(pattern)
 *
 *        while (true) {
 *          const { debounced, _action } = yield race({
 *            debounced: delay(ms),
 *            _action: take(pattern)
 *          })
 *
 *          if (debounced) {
 *            yield fork(worker, ...args, action)
 *            break
 *          }
 *
 *          action = _action
 *        }
 *      }
 *    })
 *
 * @param ms defines how many milliseconds should elapse since the last time
 *   `pattern` action was fired to call the `saga`
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `debounce` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function debounce<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function debounce<P extends ActionPattern, Fn extends (...args: any[]) => any>(
```

```
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function debounce<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect<never>
export function debounce<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `debounce(ms, pattern, saga, ...args)`.
 */
export function debounce<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function debounce<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments. In case of failure will try to make
 * another call after `delay` milliseconds, if a number of attempts < `maxTries`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `retrySaga`. We use `retry`
 * to try to fetch our API 3 times with 10 second interval. If `request` fails
 * first time than `retry` will call `request` one more time while calls count
 * less than 3.
 *
 *    import { put, retry } from 'redux-saga/effects'
 *    import { request } from 'some-api';
 *
 *    function* retrySaga(data) {
 *      try {
 *        const SECOND = 1000
 *        const response = yield retry(3, 10 * SECOND, request, data)
 *        yield put({ type: 'REQUEST_SUCCESS', payload: response })
 *      } catch(error) {
 *        yield put({ type: 'REQUEST_FAIL', payload: { error } })
 *      }
 *    }
 *
 * @param maxTries maximum calls count.
 * @param delay length of a time window in milliseconds between `fn` calls.
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as a result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
export function retry<Fn extends (...args: any[]) => any>(
  maxTries: number,
  delayLength: number,
  fn: Fn,
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>

/**
 * Creates an Effect description that instructs the middleware to run multiple
 * Effects in parallel and wait for all of them to complete. It's quite the
 * corresponding API to standard
 * [`Promise#all`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/all).
 *
 * #### Example
 *
 * The following example runs two blocking calls in parallel:
 *
 *    import { fetchCustomers, fetchProducts } from './path/to/api'
 *    import { all, call } from `redux-saga/effects`
 *
 *    function* mySaga() {
 *      const [customers, products] = yield all([
 *        call(fetchCustomers),
 *        call(fetchProducts)
 *      ])
 *    }
 */
export function all<T>(effects: T[]): AllEffect<T>
/**
 * The same as `all([...effects])` but let's you to pass in a dictionary object
 * of effects with labels, just like `race(effects)`
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function all<T>(effects: { [key: string]: T }): AllEffect<T>

export type AllEffect<T> = CombinatorEffect<'ALL', T>

export type AllEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * Creates an Effect description that instructs the middleware to run a *Race*
 * between multiple Effects (this is similar to how
 * [`Promise.race([...])`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)
 * behaves).
 *
 *
 * #### Example
 *
 * The following example runs a race between two effects:
 *
 * 1. A call to a function `fetchUsers` which returns a Promise
```

```
 * 2. A `CANCEL_FETCH` action which may be eventually dispatched on the Store
 *
 *    import { take, call, race } from `redux-saga/effects`
 *    import fetchUsers from './path/to/fetchUsers'
 *
 *    function* fetchUsersSaga() {
 *      const { response, cancel } = yield race({
 *        response: call(fetchUsers),
 *        cancel: take(CANCEL_FETCH)
 *      })
 *    }
 *
 * If `call(fetchUsers)` resolves (or rejects) first, the result of `race` will
 * be an object with a single keyed object `{response: result}` where `result`
 * is the resolved result of `fetchUsers`.
 *
 * If an action of type `CANCEL_FETCH` is dispatched on the Store before
 * `fetchUsers` completes, the result will be a single keyed object
 * `{cancel: action}`, where action is the dispatched action.
 *
 * #### Notes
 *
 * When resolving a `race`, the middleware automatically cancels all the losing
 * Effects.
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function race<T>(effects: { [key: string]: T }): RaceEffect<T>
/**
 * The same as `race(effects)` but lets you pass in an array of effects.
 */
export function race<T>(effects: T[]): RaceEffect<T>

export type RaceEffect<T> = CombinatorEffect<'RACE', T>

export type RaceEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * [H, ...T] -> T
 */
export type Tail<L extends any[]> = ((...l: L) => any) extends (h: any, ...t: infer T) => any ? T : never
/**
 * [...A, B] -> A
 */
export type AllButLast<L extends any[]> = Reverse<Tail<Reverse<L>>>
```

../redux-saga/packages/core/types/ts3.6/effects.test.ts

```
import { SagaIterator, Channel, EventChannel, MulticastChannel, Task, Buffer, END, buffers, detach } from 'redux-saga'
import {
  take,
  takeMaybe,
  put,
  putResolve,
  call,
  apply,
  cps,
  fork,
  spawn,
  join,
  cancel,
  select,
  actionChannel,
  cancelled,
  flush,
  setContext,
  getContext,
  takeEvery,
  takeLatest,
  takeLeading,
  throttle,
  delay,
  retry,
  all,
  race,
  debounce,
} from 'redux-saga/effects'
import { Action, ActionCreator } from 'redux'
import { StringableActionCreator, ActionMatchingPattern } from '@redux-saga/types'

interface MyAction extends Action {
  customField: string
}

declare const stringableActionCreator: ActionCreator<MyAction>

Object.assign(stringableActionCreator, {
  toString() {
    return 'my-action'
  },
})

const isMyAction = (action: Action): action is MyAction => {
  return action.type === 'my-action'
}

interface ChannelItem {
  someField: string
}
declare const channel: Channel<ChannelItem>
declare const eventChannel: EventChannel<ChannelItem>
declare const multicastChannel: MulticastChannel<ChannelItem>

function* testTake(): SagaIterator {
  yield take()
  yield take('my-action')
  yield take((action: Action) => action.type === 'my-action')
  yield take(isMyAction)
```

```
  // $ExpectError
  yield take(() => {})

  yield take(stringableActionCreator)

  yield take(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  // $ExpectError
  yield take([() => {}])

  yield takeMaybe(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  yield take(channel)
  yield takeMaybe(channel)

  yield take(eventChannel)
  yield takeMaybe(eventChannel)

  yield take(multicastChannel)
  yield takeMaybe(multicastChannel)

  // $ExpectError
  yield take(multicastChannel, (input: { someField: number }) => input.someField === 'foo')
  yield take(multicastChannel, (input: ChannelItem) => input.someField === 'foo')

  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield take([pattern1, pattern2])
  yield takeMaybe([pattern1, pattern2])
}

function* testPut(): SagaIterator {
  yield put({ type: 'my-action' })

  // $ExpectError
  yield put(channel, { type: 'my-action' })

  yield put(channel, { someField: '--' })
  yield put(channel, END)

  // $ExpectError
  yield put(eventChannel, { someField: '--' })
  // $ExpectError
  yield put(eventChannel, END)

  yield put(multicastChannel, { someField: '--' })
  yield put(multicastChannel, END)

  yield putResolve({ type: 'my-action' })
}

function* testCall(): SagaIterator {
  // $ExpectError
  yield call()

  // $ExpectError
  yield call({})

  yield call(() => {})

  // $ExpectError
  yield call((a: 'a') => {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // yield call(function*(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  yield call((a: 'a') => {}, 1)
  // $ExpectError
  yield call(function*(a: 'a'): SagaIterator {}, 1)
  yield call((a: 'a') => {}, 'a')
  yield call(function*(a: 'a'): SagaIterator {}, 'a')

  yield call<(a: 'a') => number>((a: 'a') => 1, 'a')

  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 1, 'b')
  yield call((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield call<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => number>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => 1,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
    'g',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: 'bar') {
      return this.foo
    },
  }

  // $ExpectError
```

```
  yield call([obj, obj.foo])
  // $ExpectError
  yield call([obj, obj.getFoo])
  yield call([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield call([obj, obj.getFoo], 1)

  // $ExpectError
  yield call([obj, 'foo'])
  // $ExpectError
  yield call([obj, 'getFoo'])
  // $ExpectError
  yield call([obj, 'getFoo'], 1)
  yield call([obj, 'getFoo'], 'bar')
  yield call<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')

  // $ExpectError
  yield call({ context: obj, fn: obj.foo })
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo })
  yield call({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield call({ context: obj, fn: 'foo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' }, 1)
  yield call({ context: obj, fn: 'getFoo' }, 'bar')
  yield call<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
}

function* testApply(): SagaIterator {
  const obj = {
    foo: 'bar',
    getFoo() {
      return this.foo
    },
    meth1(a: string) {
      return 1
    },
    meth2(a: string, b: number) {
      return 1
    },
    meth7(a: string, b: number, c: string, d: number, e: string, f: number, g: string) {
      return 1
    },
  }

  // $ExpectError
  yield apply(obj, obj.foo, [])
  yield apply(obj, obj.getFoo, [])
  yield apply<typeof obj, () => string>(obj, obj.getFoo, [])

  // $ExpectError
  yield apply(obj, 'foo', [])
  yield apply(obj, 'getFoo', [])
  yield apply<typeof obj, 'getFoo'>(obj, 'getFoo', [])

  // $ExpectError
  yield apply(obj, obj.meth1)
  // $ExpectError
  yield apply(obj, obj.meth1, [])
  // $ExpectError
  yield apply(obj, obj.meth1, [1])
  yield apply(obj, obj.meth1, ['a'])
  yield apply<typeof obj, (a: string) => number>(obj, obj.meth1, ['a'])

  // $ExpectError
  yield apply(obj, 'meth1')
  // $ExpectError
  yield apply(obj, 'meth1', [])
  // $ExpectError
  yield apply(obj, 'meth1', [1])
  yield apply(obj, 'meth1', ['a'])
  yield apply<typeof obj, 'meth1'>(obj, 'meth1', ['a'])

  // $ExpectError
  yield apply(obj, obj.meth2, ['a'])
  // $ExpectError
  yield apply(obj, obj.meth2, ['a', 'b'])
  // $ExpectError
  yield apply(obj, obj.meth2, [1, 'b'])
  yield apply(obj, obj.meth2, ['a', 1])
  yield apply<typeof obj, (a: string, b: number) => number>(obj, obj.meth2, ['a', 1])

  // $ExpectError
  yield apply(obj, 'meth2', ['a'])
  // $ExpectError
  yield apply(obj, 'meth2', ['a', 'b'])
  // $ExpectError
  yield apply(obj, 'meth2', [1, 'b'])
  yield apply(obj, 'meth2', ['a', 1])
  yield apply<typeof obj, 'meth2'>(obj, 'meth2', ['a', 1])

  // $ExpectError
  yield apply(obj, obj.meth7, [1, 'b', 'c', 'd', 'e', 'f', 'g'])
  yield apply(obj, obj.meth7, ['a', 1, 'b', 2, 'c', 3, 'd'])
  yield apply<typeof obj, (a: string, b: number, c: string, d: number, e: string, f: number, g: string) => number>(
    obj,
    obj.meth7,
    ['a', 1, 'b', 2, 'c', 3, 'd'],
  )

  // $ExpectError
  yield apply(obj, 'meth7', [1, 'b', 'c', 'd', 'e', 'f', 'g'])
  yield apply(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
  yield apply<typeof obj, 'meth7'>(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
}
```

```
function* testCps(): SagaIterator {
  type Cb<R> = (error: any, result: R) => void

  // $ExpectError
  yield cps((a: number) => {})

  // $ExpectError
  yield cps((a: number, b: string) => {}, 42)

  yield cps(cb => {
    cb(null, 1)
  })
  yield cps((cb: Cb<number>) => {
    cb(null, 1)
  })

  yield cps<(cb: Cb<string>) => void>(cb => {
    cb(null, 1)  // $ExpectError
  })
  yield cps<(cb: Cb<number>) => void>(cb => {
    cb(null, 1)
  })

  yield cps(cb => {
    cb.cancel = () => {}
  })

  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {})
  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {}, 1)
  yield cps((a: 'a', cb: Cb<number>) => {}, 'a')

  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a')
  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a', 1)
  // $ExpectError
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 1, 'b')
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 'a', 'b')

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {}, 1, 'b', 'c', 'd')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {}, 1, 'b', 'c', 'd', 'e', 'f')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: string, cb: Cb<string>) {
      cb(null, this.foo)
    },
  }
  const objWithoutCb = {
    foo: 'bar',
    getFoo(arg: string) {},
  }

  // $ExpectError
  yield cps([obj, obj.foo])
  // $ExpectError
  yield cps([obj, obj.getFoo])
  // $ExpectError
  yield cps([obj, obj.getFoo], 1)
  yield cps([obj, obj.getFoo], 'bar')
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>([obj, obj.getFoo], 'bar')
    // $ExpectError
  yield cps([objWithoutCb, objWithoutCb.getFoo])

  // $ExpectError
```

```
  yield cps([obj, 'foo'])
  // $ExpectError
  yield cps([obj, 'getFoo'])
  // $ExpectError
  yield cps([obj, 'getFoo'], 1)
  yield cps([obj, 'getFoo'], 'bar')
  yield cps<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield cps([objWithoutCb, 'getFoo'])

  // $ExpectError
  yield cps({ context: obj, fn: obj.foo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo }, 1)
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: objWithoutCb.getFoo })

  // $ExpectError
  yield cps({ context: obj, fn: 'foo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' }, 1)
  yield cps({ context: obj, fn: 'getFoo' }, 'bar')
  yield cps<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: 'getFoo' })
}

function* testFork(): SagaIterator {
  // $ExpectError
  yield fork()

  yield fork(() => {})

  // $ExpectError
  yield fork((a: 'a') => {})
  // $ExpectError
  yield fork((a: 'a') => {}, 1)
  yield fork((a: 'a') => {}, 'a')

  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 1, 'b')
  yield fork((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield fork([obj, obj.foo])
  // $ExpectError
  yield fork([obj, obj.getFoo])
  yield fork([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield fork([obj, obj.getFoo], 1)

  // $ExpectError
  yield fork([obj, 'foo'])
  // $ExpectError
  yield fork([obj, 'getFoo'])
  yield fork([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield fork([obj, 'getFoo'], 1)

  // $ExpectError
  yield fork({ context: obj, fn: obj.foo })
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo })
  yield fork({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield fork({ context: obj, fn: 'foo' })
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' })
  yield fork({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' }, 1)
}

function* testSpawn(): SagaIterator {
  // $ExpectError
  yield spawn()

  yield spawn(() => {})

  // $ExpectError
  yield spawn((a: 'a') => {})
  // $ExpectError
  yield spawn((a: 'a') => {}, 1)
  yield spawn((a: 'a') => {}, 'a')

  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
```

```
  yield spawn((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 1, 'b')
  yield spawn((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield spawn([obj, obj.foo])
  // $ExpectError
  yield spawn([obj, obj.getFoo])
  yield spawn([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield spawn([obj, obj.getFoo], 1)

  // $ExpectError
  yield spawn([obj, 'foo'])
  // $ExpectError
  yield spawn([obj, 'getFoo'])
  yield spawn([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield spawn([obj, 'getFoo'], 1)

  // $ExpectError
  yield spawn({ context: obj, fn: obj.foo })
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo })
  yield spawn({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield spawn({ context: obj, fn: 'foo' })
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' })
  yield spawn({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' }, 1)
}

declare const task: Task

function* testJoin(): SagaIterator {
  // $ExpectError
  yield join()

  // $ExpectError
  yield join({})

  yield join(task)
  // $ExpectError
  yield join(task, task)
  yield join([task, task])
  yield join([task, task, task])

  // $ExpectError
  yield join([task, task, {}])
}

function* testCancel(): SagaIterator {
  yield cancel()

  // $ExpectError
  yield cancel(undefined)
  // $ExpectError
  yield cancel({})

  yield cancel(task)
  // $ExpectError
  yield cancel(task, task)
  yield cancel([task, task])
  yield cancel([task, task, task])

  const tasks: Task[] = []

  yield cancel(tasks)

  // $ExpectError
  yield cancel([task, task, {}])
}

function* testDetach(): SagaIterator {
  yield detach(fork(() => {}))

  // $ExpectError
  yield detach(call(() => {}))
}

function* testSelect(): SagaIterator {
  interface State {
    foo: string
  }

  yield select()

  yield select((state: State) => state.foo)
  // $ExpectError
  yield select<(state: State) => number>((state: State) => state.foo)
  yield select<(state: State) => string>((state: State) => state.foo)

  // $ExpectError
  yield select((state: State, a: 'a') => state.foo)
```

```
  // $ExpectError
  yield select((state: State, a: 'a') => state.foo, 1)
  yield select((state: State, a: 'a') => state.foo, 'a')
  yield select<(state: State, a: 'a') => string>((state: State, a: 'a') => state.foo, 'a')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a')
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 1)
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 1, 'b')
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')
  yield select<(state: State, a: 'a', b: 'b') => string>((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo, 1, 'b', 'c', 'd', 'e', 'f')

  yield select(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
  yield select<(state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => string>(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
}

declare const actionBuffer: Buffer<Action>
declare const nonActionBuffer: Buffer<ChannelItem>

function* testActionChannel(): SagaIterator {
  // $ExpectError
  yield actionChannel()

  /* action type */

  yield actionChannel('my-action')
  yield actionChannel('my-action', actionBuffer)
  // $ExpectError
  yield actionChannel('my-action', nonActionBuffer)

  /* action predicate */

  yield actionChannel((action: Action) => action.type === 'my-action')
  yield actionChannel((action: Action) => action.type === 'my-action', actionBuffer)
  // $ExpectError
  yield actionChannel((action: Action) => action.type === 'my-action', nonActionBuffer)
  // $ExpectError
  yield actionChannel((item: ChannelItem) => item.someField === '--', actionBuffer)

  // $ExpectError
  yield actionChannel(() => {})
  // $ExpectError
  yield actionChannel(() => {}, actionBuffer)

  /* stringable action creator */

  yield actionChannel(stringableActionCreator)

  yield actionChannel(stringableActionCreator, buffers.fixed<MyAction>())
  // $ExpectError
  yield actionChannel(stringableActionCreator, nonActionBuffer)

  /* array */

  yield actionChannel(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator])

  // $ExpectError
  yield actionChannel([() => {}])
}

function* testCancelled(): SagaIterator {
  yield cancelled()
  // $ExpectError
  yield cancelled(1)
}

function* testFlush(): SagaIterator {
  // $ExpectError
  yield flush()
  // $ExpectError
  yield flush({})

  yield flush(channel)
  yield flush(eventChannel)
  // $ExpectError
  yield flush(multicastChannel)
}

function* testGetContext(): SagaIterator {
  // $ExpectError
  yield getContext()

  // $ExpectError
  yield getContext({})

  yield getContext('prop')
}

function* testSetContext(): SagaIterator {
  // $ExpectError
  yield setContext()
```

```
  // $ExpectError
  yield setContext('prop')

  yield setContext({ prop: 1 })
}

function* testTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery()
  // $ExpectError
  yield takeEvery('my-action')

  yield takeEvery('my-action', (action: Action) => {})
  yield takeEvery('my-action', (action: MyAction) => {})
  yield takeEvery('my-action', function*(action: Action): SagaIterator {})
  yield takeEvery('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker1)
  // $ExpectError
  yield takeEvery('my-action', helperWorker1, 1)
  yield takeEvery('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga1)
  // $ExpectError
  yield takeEvery('my-action', helperSaga1, 1)
  yield takeEvery('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const helperWorker8 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeEvery(isMyAction, action => action.customField)

  yield takeEvery(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeEvery(() => {}, (action: Action) => {})

  yield takeEvery(stringableActionCreator, action => action.customField)

  yield takeEvery(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeEvery(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeEvery([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeEvery(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
```

```
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery(channel)

  // $ExpectError
  yield takeEvery(channel, (action: Action) => {})
  yield takeEvery(channel, (action: ChannelItem) => {})
  yield takeEvery(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeEvery(channel, helperWorker1)
  // $ExpectError
  yield takeEvery(channel, helperWorker1, 1)
  yield takeEvery(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga1)
  // $ExpectError
  yield takeEvery(channel, helperSaga1, 1)
  yield takeEvery(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeEvery(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery(eventChannel, (action: ChannelItem) => {})
  yield takeEvery(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest()
  // $ExpectError
  yield takeLatest('my-action')

  yield takeLatest('my-action', (action: Action) => {})
  yield takeLatest('my-action', (action: MyAction) => {})
  yield takeLatest('my-action', function*(action: Action): SagaIterator {})
  yield takeLatest('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker1)
  // $ExpectError
  yield takeLatest('my-action', helperWorker1, 1)
  yield takeLatest('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLatest('my-action', helperSaga1)
  // $ExpectError
  yield takeLatest('my-action', helperSaga1, 1)
  yield takeLatest('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLatest(isMyAction, action => action.customField)

  yield takeLatest(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLatest(() => {}, (action: Action) => {})

  yield takeLatest(stringableActionCreator, action => action.customField)

  yield takeLatest(
```

```
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLatest(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLatest([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLatest(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest(channel)

  // $ExpectError
  yield takeLatest(channel, (action: Action) => {})
  yield takeLatest(channel, (action: ChannelItem) => {})
  yield takeLatest(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLatest(channel, helperWorker1)
  // $ExpectError
  yield takeLatest(channel, helperWorker1, 1)
  yield takeLatest(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga1)
  // $ExpectError
  yield takeLatest(channel, helperSaga1, 1)
  yield takeLatest(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeLatest(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest(eventChannel, (action: ChannelItem) => {})
  yield takeLatest(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading()
  // $ExpectError
  yield takeLeading('my-action')

  yield takeLeading('my-action', (action: Action) => {})
  yield takeLeading('my-action', (action: MyAction) => {})
  yield takeLeading('my-action', function*(action: Action): SagaIterator {})
  yield takeLeading('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker1)
  // $ExpectError
  yield takeLeading('my-action', helperWorker1, 1)
  yield takeLeading('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga1)
```

```
  // $ExpectError
  yield takeLeading('my-action', helperSaga1, 1)
  yield takeLeading('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLeading(isMyAction, action => action.customField)

  yield takeLeading(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLeading(() => {}, (action: Action) => {})

  yield takeLeading(stringableActionCreator, action => action.customField)

  yield takeLeading(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLeading(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLeading([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLeading(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading(channel)

  // $ExpectError
  yield takeLeading(channel, (action: Action) => {})
  yield takeLeading(channel, (action: ChannelItem) => {})
  yield takeLeading(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLeading(channel, helperWorker1)
  // $ExpectError
  yield takeLeading(channel, helperWorker1, 1)
  yield takeLeading(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga1)
  // $ExpectError
  yield takeLeading(channel, helperSaga1, 1)
  yield takeLeading(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}
```

```
  // $ExpectError
  yield takeLeading(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading(eventChannel, (action: ChannelItem) => {})
  yield takeLeading(multicastChannel, (action: ChannelItem) => {})
}

function* testThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1)
  // $ExpectError
  yield throttle(1, 'my-action')

  yield throttle(1, 'my-action', (action: Action) => {})
  yield throttle(1, 'my-action', (action: MyAction) => {})
  yield throttle(1, 'my-action', function*(action: Action): SagaIterator {})
  yield throttle(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1)
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1, 1)
  yield throttle(1, 'my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga1)
  // $ExpectError
  yield throttle(1, 'my-action', helperSaga1, 1)
  yield throttle(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield throttle(1, isMyAction, action => action.customField)

  yield throttle(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield throttle(1, () => {}, (action: Action) => {})

  yield throttle(1, stringableActionCreator, action => action.customField)

  yield throttle(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield throttle(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield throttle(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield throttle(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
```

```
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1, channel)

  // $ExpectError
  yield throttle(1, channel, (action: Action) => {})
  yield throttle(1, channel, (action: ChannelItem) => {})
  yield throttle(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker1)
  // $ExpectError
  yield throttle(1, channel, helperWorker1, 1)
  yield throttle(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga1)
  // $ExpectError
  yield throttle(1, channel, helperSaga1, 1)
  yield throttle(1, channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, eventChannel, (action: ChannelItem) => {})
  yield throttle(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1)
  // $ExpectError
  yield debounce(1, 'my-action')

  yield debounce(1, 'my-action', (action: Action) => {})
  yield debounce(1, 'my-action', (action: MyAction) => {})
  yield debounce(1, 'my-action', function*(action: Action): SagaIterator {})
  yield debounce(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1)
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1, 1)
  yield debounce(1, 'my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1)
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1, 1)
  yield debounce(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield debounce(1, isMyAction, action => action.customField)

  yield debounce(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield debounce(1, () => {}, (action: Action) => {})
```

```
  yield debounce(1, stringableActionCreator, action => action.customField)

  yield debounce(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield debounce(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield debounce(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield debounce(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1, channel)

  // $ExpectError
  yield debounce(1, channel, (action: Action) => {})
  yield debounce(1, channel, (action: ChannelItem) => {})
  yield debounce(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker1)
  // $ExpectError
  yield debounce(1, channel, helperWorker1, 1)
  yield debounce(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield debounce(1, channel, helperSaga1)
  // $ExpectError
  yield debounce(1, channel, helperSaga1, 1)
  yield debounce(1, channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield debounce(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, eventChannel, (action: ChannelItem) => {})
  yield debounce(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDelay(): SagaIterator {
  // $ExpectError
  yield delay()
  yield delay(1)
}

function* testRetry(): SagaIterator {
  // $ExpectError
  yield retry()
  // $ExpectError
  yield retry(1, 0, 1)
  yield retry(1, 0, () => 1)

  yield retry<() => 'foo'>(1, 0, () => 'foo')
  // $ExpectError
  yield retry<() => 'bar'>(1, 0, () => 'foo')
```

```
  yield retry(1, 0, a => a + 1, 42)
  // $ExpectError
  yield retry(1, 0, (a: string) => a, 42)
  // $ExpectError
  yield retry<(a: string) => number>(1, 0, a => a, 42)

  yield retry(1, 0, (a: number, b: number, c: string) => a, 1, 2, '3')
}

declare const promise: Promise<any>

function* testAll(): SagaIterator {
  yield all([call(() => {})])

  // $ExpectError
  yield all([1])

  // $ExpectError
  yield all([() => {}])

  // $ExpectError
  yield all([promise])

  // $ExpectError
  yield all([1, () => {}, promise])

  yield all({
    named: call(() => {}),
  })

  // $ExpectError
  yield all({
    named: 1,
  })

  // $ExpectError
  yield all({
    named: () => {},
  })

  // $ExpectError
  yield all({
    named: promise,
  })

  // $ExpectError
  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testNonStrictAll() {
  yield all([1])

  yield all([() => {}])

  yield all([promise])

  yield all([1, () => {}, promise])

  yield all({
    named: 1,
  })

  yield all({
    named: () => {},
  })

  yield all({
    named: promise,
  })

  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testRace(): SagaIterator {
  yield race({
    call: call(() => {}),
  })

  // $ExpectError
  yield race({
    named: 1,
  })

  // $ExpectError
  yield race({
    named: () => {},
  })

  // $ExpectError
  yield race({
    named: promise,
  })

  // $ExpectError
  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  // $ExpectError
```

```
  yield race([...effectArray, promise])
}

function* testNonStrictRace() {
  yield race({
    named: 1,
  })

  yield race({
    named: () => {},
  })

  yield race({
    named: promise,
  })

  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  yield race([...effectArray, promise])
}
```

../redux-saga/packages/core/types/ts3.6/index.d.ts

```
import { Saga, Buffer, Channel, END as EndType, Predicate, SagaIterator, Task, NotUndefined } from '@redux-saga/types'
import { ForkEffect } from './effects'

export { Saga, SagaIterator, Buffer, Channel, Task }

export type Action<T extends string = string> = {
  type: T
}

export interface AnyAction extends Action {
  [extraProps: string]: any
}

export interface UnknownAction extends Action {
  [extraProps: string]: unknown
}

interface Dispatch<A extends Action = UnknownAction> {
  <T extends A>(action: T, ...extraArgs: any[]): T
}

interface MiddlewareAPI<D extends Dispatch = Dispatch, S = any> {
  dispatch: D
  getState(): S
}

export interface Middleware<_DispatchExt = {}, S = any, D extends Dispatch = Dispatch> {
  (api: MiddlewareAPI<D, S>): (next: (action: never) => unknown) => (action: unknown) => unknown
}

/**
 * Used by the middleware to dispatch monitoring events. Actually the middleware
 * dispatches 6 events:
 *
 * - When a root saga is started (via `runSaga` or `sagaMiddleware.run`) the
 *   middleware invokes `sagaMonitor.rootSagaStarted`
 *
 * - When an effect is triggered (via `yield someEffect`) the middleware invokes
 *   `sagaMonitor.effectTriggered`
 *
 * - If the effect is resolved with success the middleware invokes
 *   `sagaMonitor.effectResolved`
 *
 * - If the effect is rejected with an error the middleware invokes
 *   `sagaMonitor.effectRejected`
 *
 * - If the effect is cancelled the middleware invokes
 *   `sagaMonitor.effectCancelled`
 *
 * - Finally, the middleware invokes `sagaMonitor.actionDispatched` when a Redux
 *   action is dispatched.
 */
export interface SagaMonitor {
  /**
   * @param effectId Unique ID assigned to this root saga execution
   * @param saga The generator function that starts to run
   * @param args The arguments passed to the generator function
   */
  rootSagaStarted?(options: { effectId: number; saga: Saga; args: any[] }): void
  /**
   * @param effectId Unique ID assigned to the yielded effect
   * @param parentEffectId ID of the parent Effect. In the case of a `race` or
   *   `parallel` effect, all effects yielded inside will have the direct
   *   race/parallel effect as a parent. In case of a top-level effect, the
   *   parent will be the containing Saga
   * @param label In case of a `race`/`all` effect, all child effects will be
   *   assigned as label the corresponding keys of the object passed to
   *   `race`/`all`
   * @param effect The yielded effect itself
   */
  effectTriggered?(options: { effectId: number; parentEffectId: number; label?: string; effect: any }): void
  /**
   * @param effectId The ID of the yielded effect
   * @param result The result of the successful resolution of the effect. In
   *   case of `fork` or `spawn` effects, the result will be a `Task` object.
   */
  effectResolved?(effectId: number, result: any): void
  /**
   * @param effectId The ID of the yielded effect
   * @param error Error raised with the rejection of the effect
```

```
   */
  effectRejected?(effectId: number, error: any): void
  /**
   * @param effectId The ID of the yielded effect
   */
  effectCancelled?(effectId: number): void
  /**
   * @param action The dispatched Redux action. If the action was dispatched by
   * a Saga then the action will have a property `SAGA_ACTION` set to true
   * (`SAGA_ACTION` can be imported from `@redux-saga/symbols`).
   */
  actionDispatched?(action: Action): void
}

/**
 * Creates a Redux middleware and connects the Sagas to the Redux Store
 *
 * #### Example
 *
 * Below we will create a function `configureStore` which will enhance the Store
 * with a new method `runSaga`. Then in our main module, we will use the method
 * to start the root Saga of the application.
 *
 * **configureStore.js**
 *
 *    import createSagaMiddleware from 'redux-saga'
 *    import reducer from './path/to/reducer'
 *
 *    export default function configureStore(initialState) {
 *      // Note: passing middleware as the last argument to createStore requires redux@>=3.1.0
 *      const sagaMiddleware = createSagaMiddleware()
 *      return {
 *        ...createStore(reducer, initialState, applyMiddleware(... other middleware ..., sagaMiddleware)),
 *        runSaga: sagaMiddleware.run
 *      }
 *    }
 *
 * **main.js**
 *
 *    import configureStore from './configureStore'
 *    import rootSaga from './sagas'
 *    // ... other imports
 *
 *    const store = configureStore()
 *    store.runSaga(rootSaga)
 *
 * @param options A list of options to pass to the middleware
 */
export default function createSagaMiddleware<C extends object>(options?: SagaMiddlewareOptions<C>): SagaMiddleware<C>

export interface SagaMiddlewareOptions<C extends object = {}> {
  /**
   * Initial value of the saga's context.
   */
  context?: C
  /**
   * If a Saga Monitor is provided, the middleware will deliver monitoring
   * events to the monitor.
   */
  sagaMonitor?: SagaMonitor
  /**
   * If provided, the middleware will call it with uncaught errors from Sagas.
   * useful for sending uncaught exceptions to error tracking services.
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * Allows you to intercept any effect, resolve it on your own and pass to the
   * next middleware.
   */
  effectMiddlewares?: EffectMiddleware[]
  /**
   * If provided, the middleware will use this channel instead of the default `stdChannel` for
   * take and put effects.
   */
  channel?: MulticastChannel<Action>
}

export interface SagaMiddleware<C extends object = {}> extends Middleware {
  /**
   * Dynamically run `saga`. Can be used to run Sagas **only after** the
   * `applyMiddleware` phase.
   *
   * The method returns a `Task` descriptor.
   *
   * #### Notes
   *
   * `saga` must be a function which returns a [Generator
   * Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator).
   * The middleware will then iterate over the Generator and execute all yielded
   * Effects.
   *
   * `saga` may also start other sagas using the various Effects provided by the
   * library. The iteration process described below is also applied to all child
   * sagas.
   *
   * In the first iteration, the middleware invokes the `next()` method to
   * retrieve the next Effect. The middleware then executes the yielded Effect
   * as specified by the Effects API below. Meanwhile, the Generator will be
   * suspended until the effect execution terminates. Upon receiving the result
   * of the execution, the middleware calls `next(result)` on the Generator
   * passing it the retrieved result as an argument. This process is repeated
   * until the Generator terminates normally or by throwing some error.
   *
   * If the execution results in an error (as specified by each Effect creator)
   * then the `throw(error)` method of the Generator is called instead. If the
   * Generator function defines a `try/catch` surrounding the current yield
   * instruction, then the `catch` block will be invoked by the underlying
   * Generator runtime. The runtime will also invoke any corresponding finally
   * block.
   *
   * In the case a Saga is cancelled (either manually or using the provided
   * Effects), the middleware will invoke `return()` method of the Generator.
```

```
   * This will cause the Generator to skip directly to the finally block.
   *
   * @param saga a Generator function
   * @param args arguments to be provided to `saga`
   */
  run<S extends Saga>(saga: S, ...args: Parameters<S>): Task

  setContext(props: Partial<C>): void
}

export interface EffectMiddleware {
  (next: (effect: any) => void): (effect: any) => void
}

/**
 * Allows starting sagas outside the Redux middleware environment. Useful if you
 * want to connect a Saga to external input/output, other than store actions.
 *
 * `runSaga` returns a Task object. Just like the one returned from a `fork`
 * effect.
 */
export function runSaga<Action, State, S extends Saga>(
  options: RunSagaOptions<Action, State>,
  saga: S,
  ...args: Parameters<S>
): Task

interface ErrorInfo {
  sagaStack: string
}

/**
 * The `{subscribe, dispatch}` is used to fulfill `take` and `put` Effects. This
 * defines the Input/Output interface of the Saga.
 *
 * `subscribe` is used to fulfill `take(PATTERN)` effects. It must call
 * `callback` every time it has an input to dispatch (e.g. on every mouse click
 * if the Saga is connected to DOM click events). Each time `subscribe` emits an
 * input to its callbacks, if the Saga is blocked on a `take` effect, and if the
 * take pattern matches the currently incoming input, the Saga is resumed with
 * that input.
 *
 * `dispatch` is used to fulfill `put` effects. Each time the Saga emits a
 * `yield put(output)`, `dispatch` is invoked with output.
 */
export interface RunSagaOptions<A, S> {
  /**
   * See docs for `channel`
   */
  channel?: PredicateTakeableChannel<A>
  /**
   * Used to fulfill `put` effects.
   *
   * @param output argument provided by the Saga to the `put` Effect
   */
  dispatch?(output: A): any
  /**
   * Used to fulfill `select` and `getState` effects
   */
  getState?(): S
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  sagaMonitor?: SagaMonitor
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  context?: object
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  effectMiddlewares?: EffectMiddleware[]
}

export const CANCEL: string
export const END: EndType
export type END = EndType

export interface TakeableChannel<T> {
  take(cb: (message: T | END) => void): void
}

export interface PuttableChannel<T> {
  put(message: T | END): void
}

export interface FlushableChannel<T> {
  flush(cb: (items: T[] | END) => void): void
}

/**
 * A factory method that can be used to create Channels. You can optionally pass
 * it a buffer to control how the channel buffers the messages.
 *
 * By default, if no buffer is provided, the channel will queue incoming
 * messages up to 10 until interested takers are registered. The default
 * buffering will deliver message using a FIFO strategy: a new taker will be
 * delivered the oldest message in the buffer.
 */
export function channel<T extends NotUndefined>(buffer?: Buffer<T>): Channel<T>

/**
 * Creates channel that will subscribe to an event source using the `subscribe`
 * method. Incoming events from the event source will be queued in the channel
 * until interested takers are registered.
 *
 * To notify the channel that the event source has terminated, you can notify
 * the provided subscriber with an `END`
```

```
 *
 * #### Example
 *
 * In the following example we create an event channel that will subscribe to a
 * `setInterval`
 *
 *    const countdown = (secs) => {
 *      return eventChannel(emitter => {
 *          const iv = setInterval(() => {
 *            console.log('countdown', secs)
 *            secs -= 1
 *            if (secs > 0) {
 *              emitter(secs)
 *            } else {
 *              emitter(END)
 *              clearInterval(iv)
 *              console.log('countdown terminated')
 *            }
 *          }, 1000);
 *          return () => {
 *            clearInterval(iv)
 *            console.log('countdown cancelled')
 *          }
 *        }
 *      )
 *    }
 *
 * @param subscribe used to subscribe to the underlying event source. The
 *   function must return an unsubscribe function to terminate the subscription.
 * @param buffer optional Buffer object to buffer messages on this channel. If
 *   not provided, messages will not be buffered on this channel.
 */
export function eventChannel<T extends NotUndefined>(subscribe: Subscribe<T>, buffer?: Buffer<T>): EventChannel<T>

export type Subscribe<T> = (cb: (input: T | END) => void) => Unsubscribe
export type Unsubscribe = () => void

export interface EventChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void): void
  flush(cb: (items: T[] | END) => void): void
  close(): void
}

export interface PredicateTakeableChannel<T> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
}

export interface MulticastChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
  put(message: T | END): void
  close(): void
}

export function multicastChannel<T extends NotUndefined>(): MulticastChannel<T>
export function stdChannel<T extends NotUndefined>(): MulticastChannel<T>

export function detach(forkEffect: ForkEffect): ForkEffect

/**
 * Provides some common buffers
 */
export const buffers: {
  /**
   * No buffering, new messages will be lost if there are no pending takers
   */
  none<T>(): Buffer<T>
  /**
   * New messages will be buffered up to `limit`. Overflow will raise an Error.
   * Omitting a `limit` value will result in a limit of 10.
   */
  fixed<T>(limit?: number): Buffer<T>
  /**
   * Like `fixed` but Overflow will cause the buffer to expand dynamically.
   */
  expanding<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will silently drop the messages.
   */
  dropping<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will insert the new message at the end and
   * drop the oldest message in the buffer.
   */
  sliding<T>(limit?: number): Buffer<T>
}
```

../redux-saga/packages/core/types/ts3.6/middleware.test.ts

```
import createSagaMiddleware, { SagaIterator } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'
import { applyMiddleware } from 'redux'

function testApplyMiddleware() {
  const middleware = createSagaMiddleware()

  const enhancer = applyMiddleware(middleware)
}

declare const effect: StrictEffect
declare const promise: Promise<any>

function testRun() {
  const middleware = createSagaMiddleware()

  middleware.run(function* saga(): SagaIterator {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
```

```
    // middleware.run(function* saga(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a'): SagaIterator {}, 1)

  middleware.run(function* saga(a: 'a'): SagaIterator {}, 'a')

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

  // test with any iterator i.e. when generator doesn't always yield Effects.
  middleware.run(function* saga() {
    yield promise
  })
}

function testOptions() {
  const emptyOptions = createSagaMiddleware({})

  const withOptions = createSagaMiddleware({
    onError(error) {
      console.error(error)
    },

    sagaMonitor: {
      effectTriggered() {},
    },

    effectMiddlewares: [
      next => effect => {
        setTimeout(() => {
          next(effect)
        }, 10)
      },
      next => effect => {
        setTimeout(() => {
          next(effect)
        }, 10)
      },
    ],
  })

  const withMonitor = createSagaMiddleware({
    sagaMonitor: {
      effectTriggered() {},
      effectResolved() {},
      effectRejected() {},
      effectCancelled() {},
      actionDispatched() {},
    },
  })
}

function testContext() {
  interface Context {
    a: string
    b: number
  }

  // $ExpectError
  createSagaMiddleware<Context>({ context: { c: 42 } })

  // $ExpectError
  createSagaMiddleware({ context: 42 })

  const middleware = createSagaMiddleware<Context>({
    context: { a: '', b: 42 },
  })

  // $ExpectError
  middleware.setContext({ c: 42 })

  middleware.setContext({ b: 42 })

  const task = middleware.run(function*() {
    yield effect
  })
  task.setContext({ b: 42 })

  task.setContext<Context>({ a: '' })
  // $ExpectError
  task.setContext<Context>({ c: '' })
}
```

../redux-saga/packages/core/types/ts3.6/runSaga.test.ts

```
import { SagaIterator, Task, runSaga, END, MulticastChannel } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'

declare const stdChannel: MulticastChannel<any>
declare const promise: Promise<any>
declare const effect: StrictEffect
declare const iterator: Iterator<any>

function testRunSaga() {
  const task0: Task = runSaga<{ foo: string }, { baz: boolean }, () => SagaIterator>(
    {
```

```
      context: { a: 42 },

      channel: stdChannel,

      effectMiddlewares: [
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
      ],

      getState() {
        return { baz: true }
      },

      dispatch(input) {
        input.foo
        // $ExpectError
        input.bar
      },

      sagaMonitor: {
        effectTriggered() {},
        effectResolved() {},
        effectRejected() {},
        effectCancelled() {},
        actionDispatched() {},
      },

      onError(error) {
        console.error(error)
      },
    },
    function* saga(): SagaIterator {
      yield effect
    },
  )

  // $ExpectError
  runSaga()

  // $ExpectError
  runSaga({})

  // $ExpectError
  runSaga({}, iterator)

  runSaga({}, function* saga() {
    yield effect
  })

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // runSaga({}, function* saga(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  runSaga({}, function* saga(a: 'a'): SagaIterator {}, 1)

  runSaga({}, function* saga(a: 'a'): SagaIterator {}, 'a')

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
  }

  // $ExpectError
  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

  // $ExpectError
  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

  // test with any iterator i.e. when generator doesn't always yield Effects.
  runSaga({}, function* saga() {
    yield promise
  })

  // $ExpectError
  runSaga({ context: 42 }, function* saga(): SagaIterator {})
}
```

../redux-saga/packages/core/types/ts4.2/channels.test.ts

```
import {
  buffers, Buffer, channel, Channel, EventChannel, MulticastChannel, END,
  eventChannel, multicastChannel, stdChannel,
} from "redux-saga";

function testBuffers() {
  const b1: Buffer<{foo: string}> = buffers.none<{foo: string}>();

  const b2: Buffer<{foo: string}> = buffers.dropping<{foo: string}>();
  const b3: Buffer<{foo: string}> = buffers.dropping<{foo: string}>(42);

  const b4: Buffer<{foo: string}> = buffers.expanding<{foo: string}>();
  const b5: Buffer<{foo: string}> = buffers.expanding<{foo: string}>(42);

  const b6: Buffer<{foo: string}> = buffers.fixed<{foo: string}>();
  const b7: Buffer<{foo: string}> = buffers.fixed<{foo: string}>(42);
```

```
  const b8: Buffer<{foo: string}> = buffers.sliding<{foo: string}>();
  const b9: Buffer<{foo: string}> = buffers.sliding<{foo: string}>(42);

  const buffer = buffers.none<{foo: string}>();

  // $ExpectError
  buffer.put({bar: 'bar'});
  buffer.put({foo: 'foo'});

  const isEmpty: boolean = buffer.isEmpty();

  const item = buffer.take();

  // $ExpectError
  item.foo;  // item may be undefined

  const foo: string = item!.foo;

  if (buffer.flush)
    buffer.flush();
}

function testChannel() {
  const c1: Channel<{foo: string}> = channel<{foo: string}>();
  const c2: Channel<{foo: string}> = channel(buffers.none<{foo: string}>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);

  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: Array<{bar: number}> | END) => {});
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();

  // Testing that we can't define channels that pass void or undefined
  // $ExpectError
  const voidChannel: Channel<void> = channel();
  // $ExpectError
  const voidChannel2 = channel<void>();
  // $ExpectError
  const undefinedChannel = channel<undefined>();
  // $ExpectError
  channel().put();
  // $ExpectError
  channel().put(undefined);

  // Testing that we can pass primitives into channels
  channel().put(null);
  channel().put(42);
  channel().put('test');
  channel().put(true);
}

function testEventChannel(secs: number) {
  const subscribe = (emitter: (input: number | END) => void) => {
    const iv = setInterval(() => {
      secs -= 1
      if (secs > 0) {
        emitter(secs)
      } else {
        emitter(END)
        clearInterval(iv)
      }
    }, 1000);
    return () => {
      clearInterval(iv)
    }
  };
  const c1: EventChannel<number> = eventChannel<number>(subscribe);

  const c2: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<string>()); // $ExpectError

  const c3: EventChannel<number> = eventChannel<number>(subscribe,
    buffers.none<number>());

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: string | END) => {});
  c1.take((message: number | END) => {});

  // $ExpectError
  c1.put(1);

  // $ExpectError
  c1.flush();
  // $ExpectError
  c1.flush((messages: string[] | END) => {});
  c1.flush((messages: number[] | END) => {});

  c1.close();

  // $ExpectError
  const c4: EventChannel<void> = eventChannel(() => () => {})

  // $ExpectError
  const c5 = eventChannel<void>(emit => {
    emit()
    return () => {}
```

```
  })

  const c6 = eventChannel(emit => {
    // $ExpectError
    emit()
    return () => {}
  })
}

function testMulticastChannel() {
  const c1: MulticastChannel<{foo: string}> = multicastChannel<{foo: string}>();
  const c2: MulticastChannel<{foo: string}> = stdChannel<{foo: string}>();
  // $ExpectError
  const c3: MulticastChannel<void> = stdChannel()
  // $ExpectError
  const c4 = multicastChannel<void>()
  // $ExpectError
  const c5 = stdChannel<void>()

  // $ExpectError
  c1.take();
  // $ExpectError
  c1.take((message: {bar: number} | END) => {});
  c1.take((message: {foo: string} | END) => {});

  // $ExpectError
  c1.put({bar: 1});
  c1.put({foo: 'foo'});
  c1.put(END);

  // $ExpectError
  c1.flush((messages: Array<{foo: string}> | END) => {});

  c1.close();
}
```

../redux-saga/packages/core/types/ts4.2/effects.d.ts

```
// TypeScript Version: 4.2

import {
  ActionPattern,
  Effect,
  Buffer,
  CombinatorEffect,
  CombinatorEffectDescriptor,
  SimpleEffect,
  END,
  Pattern,
  Task,
  StrictEffect,
  ActionMatchingPattern,
  SagaIterator,
} from '@redux-saga/types'

import { FlushableChannel, PuttableChannel, TakeableChannel, Action, AnyAction } from './index'

export { ActionPattern, Effect, Pattern, SimpleEffect, StrictEffect }

export const effectTypes: {
  TAKE: 'TAKE'
  PUT: 'PUT'
  ALL: 'ALL'
  RACE: 'RACE'
  CALL: 'CALL'
  CPS: 'CPS'
  FORK: 'FORK'
  JOIN: 'JOIN'
  CANCEL: 'CANCEL'
  SELECT: 'SELECT'
  ACTION_CHANNEL: 'ACTION_CHANNEL'
  CANCELLED: 'CANCELLED'
  FLUSH: 'FLUSH'
  GET_CONTEXT: 'GET_CONTEXT'
  SET_CONTEXT: 'SET_CONTEXT'
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
 * specified action on the Store. The Generator is suspended until an action
 * that matches `pattern` is dispatched.
 *
 * The result of `yield take(pattern)` is an action object being dispatched.
 *
 * `pattern` is interpreted using the following rules:
 *
 * - If `take` is called with no arguments or `'*'` all dispatched actions are
 *   matched (e.g. `take()` will match all actions)
 *
 * - If it is a function, the action is matched if `pattern(action)` is true
 *   (e.g. `take(action => action.entities)` will match all actions having a
 *   (truthy) `entities` field.)
 * > Note: if the pattern function has `toString` defined on it, `action.type`
 * > will be tested against `pattern.toString()` instead. This is useful if
 * > you're using an action creator library like redux-act or redux-actions.
 *
 * - If it is a String, the action is matched if `action.type === pattern` (e.g.
 *   `take(INCREMENT_ASYNC)`
 *
 * - If it is an array, each item in the array is matched with aforementioned
 *   rules, so the mixed array of strings and function predicates is supported.
 *   The most common use case is an array of strings though, so that
 *   `action.type` is matched against all items in the array (e.g.
 *   `take([INCREMENT, DECREMENT])` and that would match either actions of type
 *   `INCREMENT` or `DECREMENT`).
 *
 * The middleware provides a special action `END`. If you dispatch the END
 * action, then all Sagas blocked on a take Effect will be terminated regardless
 * of the specified pattern. If the terminated Saga has still some forked tasks
```

```
 * which are still running, it will wait for all the child tasks to terminate
 * before terminating the Task.
 */
export function take(pattern?: ActionPattern): TakeEffect
export function take<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

/**
 * Same as `take(pattern)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 *
 * #### Notes
 *
 * `takeMaybe` got its name from the FP analogy - it's like instead of having a
 * return type of `ACTION` (with automatic handling) we can have a type of
 * `Maybe(ACTION)` so we can handle both cases:
 *
 * - case when there is a `Just(ACTION)` (we have an action)
 * - the case of `NOTHING` (channel was closed*). i.e. we need some way to map
 *   over `END`
 *
 * internally all `dispatch`ed actions are going through the `stdChannel` which
 * is getting closed when `dispatch(END)` happens
 */
export function takeMaybe(pattern?: ActionPattern): TakeEffect
export function takeMaybe<A extends Action>(pattern?: ActionPattern<A>): TakeEffect

export type TakeEffect = SimpleEffect<'TAKE', TakeEffectDescriptor>

export interface TakeEffectDescriptor {
  pattern: ActionPattern
  maybe?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to wait for a
 * specified message from the provided Channel. If the channel is already
 * closed, then the Generator will immediately terminate following the same
 * process described above for `take(pattern)`.
 */
export function take<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

/**
 * Same as `take(channel)` but does not automatically terminate the Saga on an
 * `END` action. Instead all Sagas blocked on a take Effect will get the `END`
 * object.
 */
export function takeMaybe<T>(channel: TakeableChannel<T>, multicastPattern?: Pattern<T>): ChannelTakeEffect<T>

export type ChannelTakeEffect<T> = SimpleEffect<'TAKE', ChannelTakeEffectDescriptor<T>>

export interface ChannelTakeEffectDescriptor<T> {
  channel: TakeableChannel<T>
  pattern?: Pattern<T>
  maybe?: boolean
}

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeEvery` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action:
 *
 *    import { takeEvery } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchFetchUser() {
 *      yield takeEvery('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeEvery` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeEvery = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * `takeEvery` allows concurrent actions to be handled. In the example above,
 * when a `USER_REQUESTED` action is dispatched, a new `fetchUser` task is
 * started even if a previous `fetchUser` is still pending (for example, the
 * user clicks on a `Load User` button 2 consecutive times at a rapid rate, the
 * 2nd click will dispatch a `USER_REQUESTED` action while the `fetchUser` fired
 * on the first one hasn't yet terminated)
 *
 * `takeEvery` doesn't handle out of order responses from tasks. There is no
 * guarantee that the tasks will terminate in the same order they were started.
 * To handle out of order responses, you may consider `takeLatest` below.
 *
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeEvery` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeEvery<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function takeEvery<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
```

```
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeEvery<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeEvery<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeEvery(pattern, saga, ...args)`.
 */
export function takeEvery<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeEvery<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. And automatically cancels any previous `saga` task started
 * previously if it's still running.
 *
 * Each time an action is dispatched to the store. And if this action matches
 * `pattern`, `takeLatest` starts a new `saga` task in the background. If a
 * `saga` task was started previously (on the last action dispatched before the
 * actual action), and if this task is still running, the task will be
 * cancelled.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLatest` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action. Since `takeLatest` cancels any pending task started
 * previously, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only conclude with the latest action
 *
 *    import { takeLatest } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLatest('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLatest` is a high-level API built using `take` and `fork`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeLatest = (patternOrChannel, saga, ...args) => fork(function*() {
 *      let lastTask
 *      while (true) {
 *        const action = yield take(patternOrChannel)
 *        if (lastTask) {
 *          yield cancel(lastTask) // cancel is no-op if the task has already terminated
 *        }
 *        lastTask = yield fork(saga, ...args.concat(action))
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLatest` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function takeLatest<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function takeLatest<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeLatest<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeLatest<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLatest(pattern, saga, ...args)`.
 */
export function takeLatest<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeLatest<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on each action dispatched to the Store that matches
 * `pattern`. After spawning a task once, it blocks until spawned saga completes
 * and then starts to listen for a `pattern` again.
 *
 * In short, `takeLeading` is listening for the actions when it doesn't run a
 * saga.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchUser`. We use
 * `takeLeading` to start a new `fetchUser` task on each dispatched
 * `USER_REQUESTED` action. Since `takeLeading` ignores any new coming task
```

```
 * after it's started, we ensure that if a user triggers multiple consecutive
 * `USER_REQUESTED` actions rapidly, we'll only keep on running with the leading
 * action
 *
 *    import { takeLeading } from `redux-saga/effects`
 *
 *    function* fetchUser(action) {
 *      ...
 *    }
 *
 *    function* watchLastFetchUser() {
 *      yield takeLeading('USER_REQUESTED', fetchUser)
 *    }
 *
 * #### Notes
 *
 * `takeLeading` is a high-level API built using `take` and `call`. Here is how
 * the helper could be implemented using the low-level Effects
 *
 *    const takeLeading = (patternOrChannel, saga, ...args) => fork(function*() {
 *      while (true) {
 *        const action = yield take(patternOrChannel);
 *        yield call(saga, ...args.concat(action));
 *      }
 *    })
 *
 * @param pattern for more information see docs for [`take(pattern)`](#takepattern)
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `takeLeading` will
 *   add the incoming action to the argument list (i.e. the action will be the
 *   last argument provided to `saga`)
 */
export function takeLeading<P extends ActionPattern>(
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function takeLeading<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function takeLeading<A extends Action>(pattern: ActionPattern<A>, worker: (action: A) => any): ForkEffect<never>
export function takeLeading<A extends Action, Fn extends (...args: any[]) => any>(
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `takeLeading(pattern, saga, ...args)`.
 */
export function takeLeading<T>(channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function takeLeading<T, Fn extends (...args: any[]) => any>(
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

export type HelperWorkerParameters<T, Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends T
  ? AllButLast<Parameters<Fn>>
  : Parameters<Fn>

interface ThunkDispatch<State, ExtraThunkArg, BasicAction extends Action> {
  <ReturnType>(thunkAction: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>): ReturnType
  <Action extends BasicAction>(action: Action): Action
  <ReturnType, Action extends BasicAction>(
    action: Action | ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
  ): Action | ReturnType
}

export type ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction extends Action> = (
  dispatch: ThunkDispatch<State, ExtraThunkArg, BasicAction>,
  getState: () => State,
  extraArgument: ExtraThunkArg,
) => ReturnType

/**
 * Creates an Effect description that instructs the middleware to dispatch an
 * action to the Store. This effect is non-blocking, any errors that are
 * thrown downstream (e.g. in a reducer) will bubble back into the saga.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<A extends Action>(action: A): PutEffect<A>
export function put<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

/**
 * Just like `put` but the effect is blocking (if promise is returned from
 * `dispatch` it will wait for its resolution) and will bubble up errors from
 * downstream.
 *
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function putResolve<A extends Action>(action: A): PutEffect<A>
export function putResolve<ReturnType = any, State = any, ExtraThunkArg = any, BasicAction extends Action = Action>(
  action: ThunkAction<ReturnType, State, ExtraThunkArg, BasicAction>,
): PutEffect<BasicAction>

export type PutEffect<A extends Action = AnyAction> = SimpleEffect<'PUT', PutEffectDescriptor<A>>

export interface PutEffectDescriptor<A extends Action> {
  action: A
  channel: null
  resolve?: boolean
}

/**
 * Creates an Effect description that instructs the middleware to put an action
 * into the provided channel.
```

```
 *
 * This effect is blocking if the put is *not* buffered but immediately consumed
 * by takers. If an error is thrown in any of these takers it will bubble back
 * into the saga.
 *
 * @param channel a `Channel` Object.
 * @param action [see Redux `dispatch` documentation for complete info](https://redux.js.org/api/store#dispatchaction)
 */
export function put<T>(channel: PuttableChannel<T>, action: T | END): ChannelPutEffect<T>

export type ChannelPutEffect<T> = SimpleEffect<'PUT', ChannelPutEffectDescriptor<T>>

export interface ChannelPutEffectDescriptor<T> {
  action: T
  channel: PuttableChannel<T>
}

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments.
 *
 * #### Notes
 *
 * `fn` can be either a *normal* or a Generator function.
 *
 * The middleware invokes the function and examines its result.
 *
 * If the result is an Iterator object, the middleware will run that Generator
 * function, just like it did with the startup Generators (passed to the
 * middleware on startup). The parent Generator will be suspended until the
 * child Generator terminates normally, in which case the parent Generator is
 * resumed with the value returned by the child Generator. Or until the child
 * aborts with some error, in which case an error will be thrown inside the
 * parent Generator.
 *
 * If `fn` is a normal function and returns a Promise, the middleware will
 * suspend the Generator until the Promise is settled. After the promise is
 * resolved the Generator is resumed with the resolved value, or if the Promise
 * is rejected an error is thrown inside the Generator.
 *
 * If the result is not an Iterator object nor a Promise, the middleware will
 * immediately return that value back to the saga, so that it can resume its
 * execution synchronously.
 *
 * When an error is thrown inside the Generator, if it has a `try/catch` block
 * surrounding the current `yield` instruction, the control will be passed to
 * the `catch` block. Otherwise, the Generator aborts with the raised error, and
 * if this Generator was called by another Generator, the error will propagate
 * to the calling Generator.
 *
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
export function call<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>
/**
 * Same as `call([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield call([localStorage, 'getItem'], 'redux-saga')`
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): CallEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): CallEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `call(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>
/**
 * Same as `call([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield call({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function call<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>

export type CallEffect<RT = any> = SimpleEffect<'CALL', CallEffectDescriptor<RT>>

export interface CallEffectDescriptor<RT> {
  context: any
  fn: (...args: any[]) => SagaIterator<RT> | Promise<RT> | RT
  args: any[]
}

export type SagaReturnType<S extends Function> = S extends (...args: any[]) => SagaIterator<infer RT>
  ? RT
  : S extends (...args: any[]) => Promise<infer RT>
  ? RT
  : S extends (...args: any[]) => infer RT
  ? RT
  : never
```

```
/**
 * Alias for `call([context, fn], ...args)`.
 */
export function apply<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctx: Ctx,
  fnName: Name,
  args: Parameters<Ctx[Name]>,
): CallEffect<SagaReturnType<Ctx[Name]>>
export function apply<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctx: Ctx,
  fn: Fn,
  args: Parameters<Fn>,
): CallEffect<SagaReturnType<Fn>>

type Cast<A, B> = A extends B ? A : B
type AnyFunction = (...args: any[]) => any

type RequireCpsCallback<Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends CpsCallback<any> ? Fn : never
type RequireCpsNamedCallback<Ctx, Name extends keyof Ctx> = Last<
  Parameters<Cast<Ctx[Name], AnyFunction>>
> extends CpsCallback<any>
  ? Name
  : never

/**
 * Creates an Effect description that instructs the middleware to invoke `fn` as
 * a Node style function.
 *
 * @param fn a Node style function. i.e. a function which accepts in addition to
 *   its arguments, an additional callback to be invoked by `fn` when it
 *   terminates. The callback accepts two parameters, where the first parameter
 *   is used to report errors while the second is used to report successful
 *   results
 * @param args an array to be passed as arguments for `fn`
 */
export function cps<Fn extends (cb: CpsCallback<any>) => any>(fn: Fn): CpsEffect<ReturnType<Fn>>
export function cps<Fn extends (...args: any[]) => any>(
  fn: RequireCpsCallback<Fn>,
  ...args: CpsFunctionParameters<Fn>
): CpsEffect<ReturnType<Fn>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield cps([localStorage, 'getItem'], 'redux-saga')`
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: [Ctx, RequireCpsNamedCallback<Ctx, Name>],
  ...args: CpsFunctionParameters<Ctx[Name]>
): CpsEffect<ReturnType<Ctx[Name]>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => void }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: RequireCpsNamedCallback<Ctx, Name> },
  ...args: CpsFunctionParameters<Ctx[Name]>
): CpsEffect<ReturnType<Ctx[Name]>>
/**
 * Same as `cps(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: [Ctx, RequireCpsCallback<Fn>],
  ...args: CpsFunctionParameters<Fn>
): CpsEffect<ReturnType<Fn>>
/**
 * Same as `cps([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield cps({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function cps<Ctx, Fn extends (this: Ctx, ...args: any[]) => void>(
  ctxAndFn: { context: Ctx; fn: RequireCpsCallback<Fn> },
  ...args: CpsFunctionParameters<Fn>
): CpsEffect<ReturnType<Fn>>

export type CpsFunctionParameters<Fn extends (...args: any[]) => any> = Last<Parameters<Fn>> extends CpsCallback<any>
  ? AllButLast<Parameters<Fn>>
  : never

export interface CpsCallback<R> {
  (error: any, result: R): void
  cancel?(): void
}

export type CpsEffect<RT> = SimpleEffect<'CPS', CallEffectDescriptor<RT>>

/**
 * Creates an Effect description that instructs the middleware to perform a
 * *non-blocking call* on `fn`
 *
 * returns a `Task` object.
 *
 * #### Note
 *
 * `fork`, like `call`, can be used to invoke both normal and Generator
 * functions. But, the calls are non-blocking, the middleware doesn't suspend
 * the Generator while waiting for the result of `fn`. Instead as soon as `fn`
 * is invoked, the Generator resumes immediately.
 *
 * `fork`, alongside `race`, is a central Effect for managing concurrency
 * between Sagas.
 *
 * The result of `yield fork(fn ...args)` is a `Task` object.  An object
 * with some useful methods and properties.
 *
 * All forked tasks are *attached* to their parents. When the parent terminates
 * the execution of its own body of instructions, it will wait for all forked
 * tasks to terminate before returning.
 *
```

```
 * Errors from child tasks automatically bubble up to their parents. If any
 * forked task raises an uncaught error, then the parent task will abort with
 * the child Error, and the whole Parent's execution tree (i.e. forked tasks +
 * the *main task* represented by the parent's body if it's still running) will
 * be cancelled.
 *
 * Cancellation of a forked Task will automatically cancel all forked tasks that
 * are still executing. It'll also cancel the current Effect where the cancelled
 * task was blocked (if any).
 *
 * If a forked task fails *synchronously* (ie: fails immediately after its
 * execution before performing any async operation), then no Task is returned,
 * instead the parent will be aborted as soon as possible (since both parent and
 * child execute in parallel, the parent will abort as soon as it takes notice
 * of the child failure).
 *
 * To create *detached* forks, use `spawn` instead.
 *
 * @param fn A Generator function, or normal function which returns a Promise as result
 * @param args An array of values to be passed as arguments to `fn`
 */
export function fork<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield fork([localStorage, 'getItem'], 'redux-saga')`
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `fork(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `fork([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield fork({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function fork<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>

export type ForkEffect<RT = any> = SimpleEffect<'FORK', ForkEffectDescriptor<RT>>

export interface ForkEffectDescriptor<RT> extends CallEffectDescriptor<RT> {
  detached?: boolean
}

/**
 * Same as `fork(fn, ...args)` but creates a *detached* task. A detached task
 * remains independent from its parent and acts like a top-level task. The
 * parent will not wait for detached tasks to terminate before returning and all
 * events which may affect the parent or the detached task are completely
 * independents (error, cancellation).
 */
export function spawn<Fn extends (...args: any[]) => any>(
  fn: Fn,
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing a `fn` as string.
 * Useful for invoking object's methods, i.e.
 * `yield spawn([localStorage, 'getItem'], 'redux-saga')`
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: [Ctx, Name],
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
 */
export function spawn<Ctx extends { [P in Name]: (this: Ctx, ...args: any[]) => any }, Name extends string>(
  ctxAndFnName: { context: Ctx; fn: Name },
  ...args: Parameters<Ctx[Name]>
): ForkEffect<SagaReturnType<Ctx[Name]>>
/**
 * Same as `spawn(fn, ...args)` but supports passing a `this` context to `fn`.
 * This is useful to invoke object methods.
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: [Ctx, Fn],
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>
/**
 * Same as `spawn([context, fn], ...args)` but supports passing `context` and
 * `fn` as properties of an object, i.e.
 * `yield spawn({context: localStorage, fn: localStorage.getItem}, 'redux-saga')`.
 * `fn` can be a string or a function.
```

```
 */
export function spawn<Ctx, Fn extends (this: Ctx, ...args: any[]) => any>(
  ctxAndFn: { context: Ctx; fn: Fn },
  ...args: Parameters<Fn>
): ForkEffect<SagaReturnType<Fn>>

/**
 * Creates an Effect description that instructs the middleware to wait for the
 * result of a previously forked task.
 *
 * #### Notes
 *
 * `join` will resolve to the same outcome of the joined task (success or
 * error). If the joined task is cancelled, the cancellation will also propagate
 * to the Saga executing the join effect. Similarly, any potential callers of
 * those joiners will be cancelled as well.
 *
 * @param task A `Task` object returned by a previous `fork`
 */
export function join(task: Task): JoinEffect
/**
 * Creates an Effect description that instructs the middleware to wait for the
 * results of previously forked tasks.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function join(tasks: Task[]): JoinEffect

export type JoinEffect = SimpleEffect<'JOIN', JoinEffectDescriptor>

export type JoinEffectDescriptor = Task | Task[]

/**
 * Creates an Effect description that instructs the middleware to cancel a
 * previously forked task.
 *
 * #### Notes
 *
 * To cancel a running task, the middleware will invoke `return` on the
 * underlying Generator object. This will cancel the current Effect in the task
 * and jump to the finally block (if defined).
 *
 * Inside the finally block, you can execute any cleanup logic or dispatch some
 * action to keep the store in a consistent state (e.g. reset the state of a
 * spinner to false when an ajax request is cancelled). You can check inside the
 * finally block if a Saga was cancelled by issuing a `yield cancelled()`.
 *
 * Cancellation propagates downward to child sagas. When cancelling a task, the
 * middleware will also cancel the current Effect (where the task is currently
 * blocked). If the current Effect is a call to another Saga, it will be also
 * cancelled. When cancelling a Saga, all *attached forks* (sagas forked using
 * `yield fork()`) will be cancelled. This means that cancellation effectively
 * affects the whole execution tree that belongs to the cancelled task.
 *
 * `cancel` is a non-blocking Effect. i.e. the Saga executing it will resume
 * immediately after performing the cancellation.
 *
 * For functions which return Promise results, you can plug your own
 * cancellation logic by attaching a `[CANCEL]` to the promise.
 *
 * The following example shows how to attach cancellation logic to a Promise
 * result:
 *
 *    import { CANCEL } from 'redux-saga'
 *    import { fork, cancel } from 'redux-saga/effects'
 *
 *    function myApi() {
 *      const promise = myXhr(...)
 *
 *      promise[CANCEL] = () => myXhr.abort()
 *      return promise
 *    }
 *
 *    function* mySaga() {
 *
 *      const task = yield fork(myApi)
 *
 *      // ... later
 *      // will call promise[CANCEL] on the result of myApi
 *      yield cancel(task)
 *    }
 *
 * redux-saga will automatically cancel jqXHR objects using their `abort` method.
 *
 * @param task A `Task` object returned by a previous `fork`
 */
export function cancel(task: Task): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel
 * previously forked tasks.
 *
 * #### Notes
 *
 * It wraps the array of tasks in cancel effects, roughly becoming the
 * equivalent of `yield tasks.map(t => cancel(t))`.
 *
 * @param tasks A `Task` is the object returned by a previous `fork`
 */
export function cancel(tasks: Task[]): CancelEffect
/**
 * Creates an Effect description that instructs the middleware to cancel a task
 * in which it has been yielded (self-cancellation). It allows to reuse
 * destructor-like logic inside a `finally` blocks for both outer
 * (`cancel(task)`) and self (`cancel()`) cancellations.
 *
 * #### Example
 *
 *    function* deleteRecord({ payload }) {
 *      try {
 *        const { confirm, deny } = yield call(prompt);
 *        if (confirm) {
 *          yield put(actions.deleteRecord.confirmed())
```

```
 *        }
 *        if (deny) {
 *          yield cancel()
 *        }
 *      } catch(e) {
 *        // handle failure
 *      } finally {
 *        if (yield cancelled()) {
 *          // shared cancellation logic
 *          yield put(actions.deleteRecord.cancel(payload))
 *        }
 *      }
 *    }
 */
export function cancel(): CancelEffect

export type CancelEffect = SimpleEffect<'CANCEL', CancelEffectDescriptor>

export type CancelEffectDescriptor = Task | Task[] | SELF_CANCELLATION
type SELF_CANCELLATION = '@@redux-saga/SELF_CANCELLATION'

/**
 * Creates an effect that instructs the middleware to invoke the provided
 * selector on the current Store's state (i.e. returns the result of
 * `selector(getState(), ...args)`).
 *
 * If `select` is called without argument (i.e. `yield select()`) then the
 * effect is resolved with the entire state (the same result of a `getState()`
 * call).
 *
 * > It's important to note that when an action is dispatched to the store, the
 * middleware first forwards the action to the reducers and then notifies the
 * Sagas. This means that when you query the Store's State, you get the State
 * **after** the action has been applied. However, this behavior is only
 * guaranteed if all subsequent middlewares call `next(action)` synchronously.
 * If any subsequent middleware calls `next(action)` asynchronously (which is
 * unusual but possible), then the sagas will get the state from **before** the
 * action is applied.  Therefore it is recommended to review the source of each
 * subsequent middleware to ensure it calls `next(action)` synchronously, or
 * else ensure that redux-saga is the last middleware in the call chain.
 *
 * #### Notes
 *
 * Preferably, a Saga should be autonomous and should not depend on the Store's
 * state. This makes it easy to modify the state implementation without
 * affecting the Saga code. A saga should preferably depend only on its own
 * internal control state when possible. But sometimes, one could find it more
 * convenient for a Saga to query the state instead of maintaining the needed
 * data by itself (for example, when a Saga duplicates the logic of invoking
 * some reducer to compute a state that was already computed by the Store).
 *
 * For example, suppose we have this state shape in our application:
 *
 *    state = {
 *      cart: {...}
 *    }
 *
 * We can create a *selector*, i.e. a function which knows how to extract the
 * `cart` data from the State:
 *
 * `./selectors`
 *
 *    export const getCart = state => state.cart
 *
 *
 * Then we can use that selector from inside a Saga using the `select` Effect:
 *
 * `./sagas.js`
 *
 *    import { take, fork, select } from 'redux-saga/effects'
 *    import { getCart } from './selectors'
 *
 *    function* checkout() {
 *      // query the state using the exported selector
 *      const cart = yield select(getCart)
 *
 *      // ... call some API endpoint then dispatch a success/error action
 *    }
 *
 *    export default function* rootSaga() {
 *      while (true) {
 *        yield take('CHECKOUT_REQUEST')
 *        yield fork(checkout)
 *      }
 *    }
 *
 * `checkout` can get the needed information directly by using
 * `select(getCart)`. The Saga is coupled only with the `getCart` selector. If
 * we have many Sagas (or React Components) that needs to access the `cart`
 * slice, they will all be coupled to the same function `getCart`. And if we now
 * change the state shape, we need only to update `getCart`.
 *
 * @param selector a function `(state, ...args) => args`. It takes the current
 *   state and optionally some arguments and returns a slice of the current
 *   Store's state
 * @param args optional arguments to be passed to the selector in addition of
 *    `getState`.
 */
export function select(): SelectEffect
export function select<Fn extends (state: any, ...args: any[]) => any>(
  selector: Fn,
  ...args: Tail<Parameters<Fn>>
): SelectEffect

export type SelectEffect = SimpleEffect<'SELECT', SelectEffectDescriptor>

export interface SelectEffectDescriptor {
  selector(state: any, ...args: any[]): any
  args: any[]
}

/**
```

```
 * Creates an effect that instructs the middleware to queue the actions matching
 * `pattern` using an event channel. Optionally, you can provide a buffer to
 * control buffering of the queued actions.
 *
 * #### Example
 *
 * The following code creates a channel to buffer all `USER_REQUEST` actions.
 * Note that even the Saga may be blocked on the `call` effect. All actions that
 * come while it's blocked are automatically buffered. This causes the Saga to
 * execute the API calls one at a time
 *
 *    import { actionChannel, call } from 'redux-saga/effects'
 *    import api from '...'
 *
 *    function* takeOneAtMost() {
 *      const chan = yield actionChannel('USER_REQUEST')
 *      while (true) {
 *        const {payload} = yield take(chan)
 *        yield call(api.getUser, payload)
 *      }
 *    }
 *
 * @param pattern see API for `take(pattern)`
 * @param buffer a `Buffer` object
 */
export function actionChannel(pattern: ActionPattern, buffer?: Buffer<Action>): ActionChannelEffect

export type ActionChannelEffect = SimpleEffect<'ACTION_CHANNEL', ActionChannelEffectDescriptor>

export interface ActionChannelEffectDescriptor {
  pattern: ActionPattern
  buffer?: Buffer<Action>
}

/**
 * Creates an effect that instructs the middleware to flush all buffered items
 * from the channel. Flushed items are returned back to the saga, so they can be
 * utilized if needed.
 *
 * #### Example
 *
 *    function* saga() {
 *      const chan = yield actionChannel('ACTION')
 *
 *      try {
 *        while (true) {
 *          const action = yield take(chan)
 *          // ...
 *        }
 *      } finally {
 *        const actions = yield flush(chan)
 *        // ...
 *      }
 *    }
 *
 * @param channel a `Channel` Object.
 */
export function flush<T>(channel: FlushableChannel<T>): FlushEffect<T>

export type FlushEffect<T> = SimpleEffect<'FLUSH', FlushEffectDescriptor<T>>

export type FlushEffectDescriptor<T> = FlushableChannel<T>

/**
 * Creates an effect that instructs the middleware to return whether this
 * generator has been cancelled. Typically you use this Effect in a finally
 * block to run Cancellation specific code
 *
 * #### Example
 *
 *    function* saga() {
 *      try {
 *        // ...
 *      } finally {
 *        if (yield cancelled()) {
 *          // logic that should execute only on Cancellation
 *        }
 *        // logic that should execute in all situations (e.g. closing a channel)
 *      }
 *    }
 */
export function cancelled(): CancelledEffect

export type CancelledEffect = SimpleEffect<'CANCELLED', CancelledEffectDescriptor>

export type CancelledEffectDescriptor = {}

/**
 * Creates an effect that instructs the middleware to update its own context.
 * This effect extends saga's context instead of replacing it.
 */
export function setContext<C extends object>(props: C): SetContextEffect<C>

export type SetContextEffect<C extends object> = SimpleEffect<'SET_CONTEXT', SetContextEffectDescriptor<C>>

export type SetContextEffectDescriptor<C extends object> = C

/**
 * Creates an effect that instructs the middleware to return a specific property
 * of saga's context.
 */
export function getContext(prop: string): GetContextEffect

export type GetContextEffect = SimpleEffect<'GET_CONTEXT', GetContextEffectDescriptor>

export type GetContextEffectDescriptor = string

/**
 * Returns an effect descriptor to block execution for `ms` milliseconds and return `val` value.
 */
export function delay<T = true>(ms: number, val?: T): CallEffect<T>
```

```
/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * After spawning a task it's still accepting incoming actions into the
 * underlying `buffer`, keeping at most 1 (the most recent one), but in the same
 * time holding up with spawning new task for `ms` milliseconds (hence its name -
 * `throttle`). Purpose of this is to ignore incoming actions for a given
 * period of time while processing a task.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `throttle` to start a new `fetchAutocomplete` task on dispatched
 * `FETCH_AUTOCOMPLETE` action. However since `throttle` ignores consecutive
 * `FETCH_AUTOCOMPLETE` for some time, we ensure that user won't flood our
 * server with requests.
 *
 *    import { call, put, throttle } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* throttleAutocomplete() {
 *      yield throttle(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `throttle` is a high-level API built using `take`, `fork` and
 * `actionChannel`. Here is how the helper could be implemented using the
 * low-level Effects
 *
 *    const throttle = (ms, pattern, task, ...args) => fork(function*() {
 *      const throttleChannel = yield actionChannel(pattern, buffers.sliding(1))
 *
 *      while (true) {
 *        const action = yield take(throttleChannel)
 *        yield fork(task, ...args, action)
 *        yield delay(ms)
 *      }
 *    })
 *
 * @param ms length of a time window in milliseconds during which actions will
 *   be ignored after the action starts processing
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `throttle` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function throttle<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function throttle<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function throttle<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect<never>
export function throttle<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `throttle(ms, pattern, saga, ...args)`.
 */
export function throttle<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function throttle<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Spawns a `saga` on an action dispatched to the Store that matches `pattern`.
 * Saga will be called after it stops taking `pattern` actions for `ms`
 * milliseconds. Purpose of this is to prevent calling saga until the actions
 * are settled off.
 *
 * #### Example
 *
 * In the following example, we create a basic task `fetchAutocomplete`. We use
 * `debounce` to delay calling `fetchAutocomplete` saga until we stop receive
 * any `FETCH_AUTOCOMPLETE` events for at least `1000` ms.
 *
 *    import { call, put, debounce } from `redux-saga/effects`
 *
 *    function* fetchAutocomplete(action) {
 *      const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
 *      yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals})
 *    }
 *
 *    function* debounceAutocomplete() {
 *      yield debounce(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
 *    }
 *
 * #### Notes
 *
 * `debounce` is a high-level API built using `take`, `delay` and `fork`. Here
```

```
 * is how the helper could be implemented using the low-level Effects
 *
 *    const debounce = (ms, pattern, task, ...args) => fork(function*() {
 *      while (true) {
 *        let action = yield take(pattern)
 *
 *        while (true) {
 *          const { debounced, _action } = yield race({
 *            debounced: delay(ms),
 *            _action: take(pattern)
 *          })
 *
 *          if (debounced) {
 *            yield fork(worker, ...args, action)
 *            break
 *          }
 *
 *          action = _action
 *        }
 *      }
 *    })
 *
 * @param ms defines how many milliseconds should elapse since the last time
 *   `pattern` action was fired to call the `saga`
 * @param pattern for more information see docs for `take(pattern)`
 * @param saga a Generator function
 * @param args arguments to be passed to the started task. `debounce` will add
 *   the incoming action to the argument list (i.e. the action will be the last
 *   argument provided to `saga`)
 */
export function debounce<P extends ActionPattern>(
  ms: number,
  pattern: P,
  worker: (action: ActionMatchingPattern<P>) => any,
): ForkEffect<never>
export function debounce<P extends ActionPattern, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: P,
  worker: Fn,
  ...args: HelperWorkerParameters<ActionMatchingPattern<P>, Fn>
): ForkEffect<never>
export function debounce<A extends Action>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: (action: A) => any,
): ForkEffect<never>
export function debounce<A extends Action, Fn extends (...args: any[]) => any>(
  ms: number,
  pattern: ActionPattern<A>,
  worker: Fn,
  ...args: HelperWorkerParameters<A, Fn>
): ForkEffect<never>

/**
 * You can also pass in a channel as argument and the behaviour is the same as
 * `debounce(ms, pattern, saga, ...args)`.
 */
export function debounce<T>(ms: number, channel: TakeableChannel<T>, worker: (item: T) => any): ForkEffect<never>
export function debounce<T, Fn extends (...args: any[]) => any>(
  ms: number,
  channel: TakeableChannel<T>,
  worker: Fn,
  ...args: HelperWorkerParameters<T, Fn>
): ForkEffect<never>

/**
 * Creates an Effect description that instructs the middleware to call the
 * function `fn` with `args` as arguments. In case of failure will try to make
 * another call after `delay` milliseconds, if a number of attempts < `maxTries`.
 *
 * #### Example
 *
 * In the following example, we create a basic task `retrySaga`. We use `retry`
 * to try to fetch our API 3 times with 10 second interval. If `request` fails
 * first time than `retry` will call `request` one more time while calls count
 * less than 3.
 *
 *    import { put, retry } from 'redux-saga/effects'
 *    import { request } from 'some-api';
 *
 *    function* retrySaga(data) {
 *      try {
 *        const SECOND = 1000
 *        const response = yield retry(3, 10 * SECOND, request, data)
 *        yield put({ type: 'REQUEST_SUCCESS', payload: response })
 *      } catch(error) {
 *        yield put({ type: 'REQUEST_FAIL', payload: { error } })
 *      }
 *    }
 *
 * @param maxTries maximum calls count.
 * @param delay length of a time window in milliseconds between `fn` calls.
 * @param fn A Generator function, or normal function which either returns a
 *   Promise as a result, or any other value.
 * @param args An array of values to be passed as arguments to `fn`
 */
export function retry<Fn extends (...args: any[]) => any>(
  maxTries: number,
  delayLength: number,
  fn: Fn,
  ...args: Parameters<Fn>
): CallEffect<SagaReturnType<Fn>>

/**
 * Creates an Effect description that instructs the middleware to run multiple
 * Effects in parallel and wait for all of them to complete. It's quite the
 * corresponding API to standard
 * [`Promise#all`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/all).
 *
 * #### Example
 *
 * The following example runs two blocking calls in parallel:
```

```
 *
 *    import { fetchCustomers, fetchProducts } from './path/to/api'
 *    import { all, call } from `redux-saga/effects`
 *
 *    function* mySaga() {
 *      const [customers, products] = yield all([
 *        call(fetchCustomers),
 *        call(fetchProducts)
 *      ])
 *    }
 */
export function all<T>(effects: T[]): AllEffect<T>
/**
 * The same as `all([...effects])` but let's you to pass in a dictionary object
 * of effects with labels, just like `race(effects)`
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function all<T>(effects: { [key: string]: T }): AllEffect<T>

export type AllEffect<T> = CombinatorEffect<'ALL', T>

export type AllEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * Creates an Effect description that instructs the middleware to run a *Race*
 * between multiple Effects (this is similar to how
 * [`Promise.race([...])`](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)
 * behaves).
 *
 *
 * #### Example
 *
 * The following example runs a race between two effects:
 *
 * 1. A call to a function `fetchUsers` which returns a Promise
 * 2. A `CANCEL_FETCH` action which may be eventually dispatched on the Store
 *
 *
 *    import { take, call, race } from `redux-saga/effects`
 *    import fetchUsers from './path/to/fetchUsers'
 *
 *    function* fetchUsersSaga() {
 *      const { response, cancel } = yield race({
 *        response: call(fetchUsers),
 *        cancel: take(CANCEL_FETCH)
 *      })
 *    }
 *
 * If `call(fetchUsers)` resolves (or rejects) first, the result of `race` will
 * be an object with a single keyed object `{response: result}` where `result`
 * is the resolved result of `fetchUsers`.
 *
 * If an action of type `CANCEL_FETCH` is dispatched on the Store before
 * `fetchUsers` completes, the result will be a single keyed object
 * `{cancel: action}`, where action is the dispatched action.
 *
 * #### Notes
 *
 * When resolving a `race`, the middleware automatically cancels all the losing
 * Effects.
 *
 * @param effects a dictionary Object of the form {label: effect, ...}
 */
export function race<T>(effects: { [key: string]: T }): RaceEffect<T>
/**
 * The same as `race(effects)` but lets you pass in an array of effects.
 */
export function race<T>(effects: T[]): RaceEffect<T>

export type RaceEffect<T> = CombinatorEffect<'RACE', T>

export type RaceEffectDescriptor<T> = CombinatorEffectDescriptor<T>

/**
 * [H, ...T] -> T
 */
export type Tail<L extends any[]> = ((...l: L) => any) extends (h: any, ...t: infer T) => any ? T : never
/**
 * [...A, B] -> A
 */
export type AllButLast<L extends any[]> = L extends [] ? [] : L extends [...infer R, infer _] ? R : L

type Last<L extends any[]> = L extends [] ? never : L extends [...infer _, infer R] ? R : L[number]
```

../redux-saga/packages/core/types/ts4.2/effects.test.ts

```
import { SagaIterator, Channel, EventChannel, MulticastChannel, Task, Buffer, END, buffers, detach } from 'redux-saga'
import {
  take,
  takeMaybe,
  put,
  putResolve,
  call,
  apply,
  cps,
  fork,
  spawn,
  join,
  cancel,
  select,
  actionChannel,
  cancelled,
  flush,
  setContext,
  getContext,
  takeEvery,
  takeLatest,
  takeLeading,
```

```
  throttle,
  delay,
  retry,
  all,
  race,
  debounce,
} from 'redux-saga/effects'
import { Action, ActionCreator } from 'redux'
import { StringableActionCreator, ActionMatchingPattern } from '@redux-saga/types'

interface MyAction extends Action {
  customField: string
}

declare const stringableActionCreator: ActionCreator<MyAction>

Object.assign(stringableActionCreator, {
  toString() {
    return 'my-action'
  },
})

const isMyAction = (action: Action): action is MyAction => {
  return action.type === 'my-action'
}

interface ChannelItem {
  someField: string
}
declare const channel: Channel<ChannelItem>
declare const eventChannel: EventChannel<ChannelItem>
declare const multicastChannel: MulticastChannel<ChannelItem>

function* testTake(): SagaIterator {
  yield take()
  yield take('my-action')
  yield take((action: Action) => action.type === 'my-action')
  yield take(isMyAction)

  // $ExpectError
  yield take(() => {})

  yield take(stringableActionCreator)

  yield take(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  // $ExpectError
  yield take([() => {}])

  yield takeMaybe(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction])

  yield take(channel)
  yield takeMaybe(channel)

  yield take(eventChannel)
  yield takeMaybe(eventChannel)

  yield take(multicastChannel)
  yield takeMaybe(multicastChannel)

  // $ExpectError
  yield take(multicastChannel, (input: { someField: number }) => input.someField === 'foo')
  yield take(multicastChannel, (input: ChannelItem) => input.someField === 'foo')

  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield take([pattern1, pattern2])
  yield takeMaybe([pattern1, pattern2])
}

function* testPut(): SagaIterator {
  yield put({ type: 'my-action' })

  // $ExpectError
  yield put(channel, { type: 'my-action' })

  yield put(channel, { someField: '--' })
  yield put(channel, END)

  // $ExpectError
  yield put(eventChannel, { someField: '--' })
  // $ExpectError
  yield put(eventChannel, END)

  yield put(multicastChannel, { someField: '--' })
  yield put(multicastChannel, END)

  yield putResolve({ type: 'my-action' })
}

function* testCall(): SagaIterator {
  // $ExpectError
  yield call()

  // $ExpectError
  yield call({})

  yield call(() => {})

  // $ExpectError
  yield call((a: 'a') => {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // yield call(function*(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  yield call((a: 'a') => {}, 1)
  // $ExpectError
```

```
  yield call(function*(a: 'a'): SagaIterator {}, 1)
  yield call((a: 'a') => {}, 'a')
  yield call(function*(a: 'a'): SagaIterator {}, 'a')

  yield call<(a: 'a') => number>((a: 'a') => 1, 'a')

  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield call((a: 'a', b: 'b') => {}, 1, 'b')
  yield call((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield call((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield call<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => number>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => 1,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
    'g',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: 'bar') {
      return this.foo
    },
  }

  // $ExpectError
  yield call([obj, obj.foo])
  // $ExpectError
  yield call([obj, obj.getFoo])
  yield call([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield call([obj, obj.getFoo], 1)

  // $ExpectError
  yield call([obj, 'foo'])
  // $ExpectError
  yield call([obj, 'getFoo'])
  // $ExpectError
  yield call([obj, 'getFoo'], 1)
  yield call([obj, 'getFoo'], 'bar')
  yield call<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')

  // $ExpectError
  yield call({ context: obj, fn: obj.foo })
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo })
  yield call({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield call({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield call({ context: obj, fn: 'foo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield call({ context: obj, fn: 'getFoo' }, 1)
  yield call({ context: obj, fn: 'getFoo' }, 'bar')
  yield call<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
}

function* testApply(): SagaIterator {
  const obj = {
    foo: 'bar',
    getFoo() {
      return this.foo
    },
    meth1(a: string) {
      return 1
    },
    meth2(a: string, b: number) {
      return 1
    },
    meth7(a: string, b: number, c: string, d: number, e: string, f: number, g: string) {
      return 1
    },
  }

  // $ExpectError
  yield apply(obj, obj.foo, [])
  yield apply(obj, obj.getFoo, [])
  yield apply<typeof obj, () => string>(obj, obj.getFoo, [])

  // $ExpectError
  yield apply(obj, 'foo', [])
  yield apply(obj, 'getFoo', [])
  yield apply<typeof obj, 'getFoo'>(obj, 'getFoo', [])

  // $ExpectError
  yield apply(obj, obj.meth1)
  // $ExpectError
  yield apply(obj, obj.meth1, [])
  // $ExpectError
  yield apply(obj, obj.meth1, [1])
  yield apply(obj, obj.meth1, ['a'])
  yield apply<typeof obj, (a: string) => number>(obj, obj.meth1, ['a'])

  // $ExpectError
  yield apply(obj, 'meth1')
  // $ExpectError
  yield apply(obj, 'meth1', [])
```

```
  // $ExpectError
  yield apply(obj, 'meth1', [1])
  yield apply(obj, 'meth1', ['a'])
  yield apply<typeof obj, 'meth1'>(obj, 'meth1', ['a'])

  // $ExpectError
  yield apply(obj, obj.meth2, ['a'])
  // $ExpectError
  yield apply(obj, obj.meth2, ['a', 'b'])
  // $ExpectError
  yield apply(obj, obj.meth2, [1, 'b'])
  yield apply(obj, obj.meth2, ['a', 1])
  yield apply<typeof obj, (a: string, b: number) => number>(obj, obj.meth2, ['a', 1])

  // $ExpectError
  yield apply(obj, 'meth2', ['a'])
  // $ExpectError
  yield apply(obj, 'meth2', ['a', 'b'])
  // $ExpectError
  yield apply(obj, 'meth2', [1, 'b'])
  yield apply(obj, 'meth2', ['a', 1])
  yield apply<typeof obj, 'meth2'>(obj, 'meth2', ['a', 1])

  // $ExpectError
  yield apply(obj, obj.meth7, [1, 'b', 'c', 'd', 'e', 'f', 'g'])
  yield apply(obj, obj.meth7, ['a', 1, 'b', 2, 'c', 3, 'd'])
  yield apply<typeof obj, (a: string, b: number, c: string, d: number, e: string, f: number, g: string) => number>(
    obj,
    obj.meth7,
    ['a', 1, 'b', 2, 'c', 3, 'd'],
  )

  // $ExpectError
  yield apply(obj, 'meth7', [1, 'b', 'c', 'd', 'e', 'f', 'g'])
  yield apply(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
  yield apply<typeof obj, 'meth7'>(obj, 'meth7', ['a', 1, 'b', 2, 'c', 3, 'd'])
}

function* testCps(): SagaIterator {
  type Cb<R> = (error: any, result: R) => void

  // $ExpectError
  yield cps((a: number) => {})

  // $ExpectError
  yield cps((a: number, b: string) => {}, 42)

  yield cps(cb => {
    cb(null, 1)
  })
  yield cps((cb: Cb<number>) => {
    cb(null, 1)
  })

  yield cps<(cb: Cb<string>) => void>(cb => {
    cb(null, 1)  // $ExpectError
  })
  yield cps<(cb: Cb<number>) => void>(cb => {
    cb(null, 1)
  })

  yield cps(cb => {
    cb.cancel = () => {}
  })

  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {})
  // $ExpectError
  yield cps((a: 'a', cb: Cb<number>) => {}, 1)
  yield cps((a: 'a', cb: Cb<number>) => {}, 'a')

  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a')
  // $ExpectError
  yield cps((a: 'a', b: 'b', cb) => {}, 'a', 1)
  // $ExpectError
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 1, 'b')
  yield cps((a: 'a', b: 'b', cb: Cb<number>) => {}, 'a', 'b')

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {}, 1, 'b', 'c', 'd')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
  )

  // $ExpectError
  yield cps((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {}, 1, 'b', 'c', 'd', 'e', 'f')

  yield cps(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
```

```
    'e',
    'f',
  )
  yield cps<(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => void>(
    (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', cb: Cb<number>) => {
      cb(null, 1)
    },
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )

  const obj = {
    foo: 'bar',
    getFoo(arg: string, cb: Cb<string>) {
      cb(null, this.foo)
    },
  }
  const objWithoutCb = {
    foo: 'bar',
    getFoo(arg: string) {},
  }

  // $ExpectError
  yield cps([obj, obj.foo])
  // $ExpectError
  yield cps([obj, obj.getFoo])
  // $ExpectError
  yield cps([obj, obj.getFoo], 1)
  yield cps([obj, obj.getFoo], 'bar')
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>([obj, obj.getFoo], 'bar')
    // $ExpectError
  yield cps([objWithoutCb, objWithoutCb.getFoo])

  // $ExpectError
  yield cps([obj, 'foo'])
  // $ExpectError
  yield cps([obj, 'getFoo'])
  // $ExpectError
  yield cps([obj, 'getFoo'], 1)
  yield cps([obj, 'getFoo'], 'bar')
  yield cps<typeof obj, 'getFoo'>([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield cps([objWithoutCb, 'getFoo'])

  // $ExpectError
  yield cps({ context: obj, fn: obj.foo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo })
  // $ExpectError
  yield cps({ context: obj, fn: obj.getFoo }, 1)
  yield cps<typeof obj, (arg: string, cb: Cb<string>) => void>({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: objWithoutCb.getFoo })

  // $ExpectError
  yield cps({ context: obj, fn: 'foo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' })
  // $ExpectError
  yield cps({ context: obj, fn: 'getFoo' }, 1)
  yield cps({ context: obj, fn: 'getFoo' }, 'bar')
  yield cps<typeof obj, 'getFoo'>({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield cps({ context: objWithoutCb, fn: 'getFoo' })
}

function* testFork(): SagaIterator {
  // $ExpectError
  yield fork()

  yield fork(() => {})

  // $ExpectError
  yield fork((a: 'a') => {})
  // $ExpectError
  yield fork((a: 'a') => {}, 1)
  yield fork((a: 'a') => {}, 'a')

  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield fork((a: 'a', b: 'b') => {}, 1, 'b')
  yield fork((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield fork((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield fork([obj, obj.foo])
  // $ExpectError
  yield fork([obj, obj.getFoo])
  yield fork([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield fork([obj, obj.getFoo], 1)

  // $ExpectError
  yield fork([obj, 'foo'])
```

```
  // $ExpectError
  yield fork([obj, 'getFoo'])
  yield fork([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield fork([obj, 'getFoo'], 1)

  // $ExpectError
  yield fork({ context: obj, fn: obj.foo })
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo })
  yield fork({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield fork({ context: obj, fn: 'foo' })
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' })
  yield fork({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield fork({ context: obj, fn: 'getFoo' }, 1)
}

function* testSpawn(): SagaIterator {
  // $ExpectError
  yield spawn()

  yield spawn(() => {})

  // $ExpectError
  yield spawn((a: 'a') => {})
  // $ExpectError
  yield spawn((a: 'a') => {}, 1)
  yield spawn((a: 'a') => {}, 'a')

  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 'a')
  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 'a', 1)
  // $ExpectError
  yield spawn((a: 'a', b: 'b') => {}, 1, 'b')
  yield spawn((a: 'a', b: 'b') => {}, 'a', 'b')

  // $ExpectError
  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 1, 'b', 'c', 'd', 'e', 'f', 'g')

  yield spawn((a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const obj = {
    foo: 'bar',
    getFoo(arg: string) {
      return this.foo
    },
  }

  // $ExpectError
  yield spawn([obj, obj.foo])
  // $ExpectError
  yield spawn([obj, obj.getFoo])
  yield spawn([obj, obj.getFoo], 'bar')
  // $ExpectError
  yield spawn([obj, obj.getFoo], 1)

  // $ExpectError
  yield spawn([obj, 'foo'])
  // $ExpectError
  yield spawn([obj, 'getFoo'])
  yield spawn([obj, 'getFoo'], 'bar')
  // $ExpectError
  yield spawn([obj, 'getFoo'], 1)

  // $ExpectError
  yield spawn({ context: obj, fn: obj.foo })
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo })
  yield spawn({ context: obj, fn: obj.getFoo }, 'bar')
  // $ExpectError
  yield spawn({ context: obj, fn: obj.getFoo }, 1)

  // $ExpectError
  yield spawn({ context: obj, fn: 'foo' })
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' })
  yield spawn({ context: obj, fn: 'getFoo' }, 'bar')
  // $ExpectError
  yield spawn({ context: obj, fn: 'getFoo' }, 1)
}

declare const task: Task

function* testJoin(): SagaIterator {
  // $ExpectError
  yield join()

  // $ExpectError
  yield join({})

  yield join(task)
  // $ExpectError
  yield join(task, task)
  yield join([task, task])
  yield join([task, task, task])

  // $ExpectError
  yield join([task, task, {}])
}

function* testCancel(): SagaIterator {
  yield cancel()

  // $ExpectError
  yield cancel(undefined)
```

```
  // $ExpectError
  yield cancel({})

  yield cancel(task)
  // $ExpectError
  yield cancel(task, task)
  yield cancel([task, task])
  yield cancel([task, task, task])

  const tasks: Task[] = []

  yield cancel(tasks)

  // $ExpectError
  yield cancel([task, task, {}])
}

function* testDetach(): SagaIterator {
  yield detach(fork(() => {}))

  // $ExpectError
  yield detach(call(() => {}))
}

function* testSelect(): SagaIterator {
  interface State {
    foo: string
  }

  yield select()

  yield select((state: State) => state.foo)
  // $ExpectError
  yield select<(state: State) => number>((state: State) => state.foo)
  yield select<(state: State) => string>((state: State) => state.foo)

  // $ExpectError
  yield select((state: State, a: 'a') => state.foo)
  // $ExpectError
  yield select((state: State, a: 'a') => state.foo, 1)
  yield select((state: State, a: 'a') => state.foo, 'a')
  yield select<(state: State, a: 'a') => string>((state: State, a: 'a') => state.foo, 'a')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a')
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 1)
  // $ExpectError
  yield select((state: State, a: 'a', b: 'b') => state.foo, 1, 'b')
  yield select((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')
  yield select<(state: State, a: 'a', b: 'b') => string>((state: State, a: 'a', b: 'b') => state.foo, 'a', 'b')

  // $ExpectError
  yield select((state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo, 1, 'b', 'c', 'd', 'e', 'f')

  yield select(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
  yield select<(state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => string>(
    (state: State, a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f') => state.foo,
    'a',
    'b',
    'c',
    'd',
    'e',
    'f',
  )
}

declare const actionBuffer: Buffer<Action>
declare const nonActionBuffer: Buffer<ChannelItem>

function* testActionChannel(): SagaIterator {
  // $ExpectError
  yield actionChannel()

  /* action type */

  yield actionChannel('my-action')
  yield actionChannel('my-action', actionBuffer)
  // $ExpectError
  yield actionChannel('my-action', nonActionBuffer)

  /* action predicate */

  yield actionChannel((action: Action) => action.type === 'my-action')
  yield actionChannel((action: Action) => action.type === 'my-action', actionBuffer)
  // $ExpectError
  yield actionChannel((action: Action) => action.type === 'my-action', nonActionBuffer)
  // $ExpectError
  yield actionChannel((item: ChannelItem) => item.someField === '--', actionBuffer)

  // $ExpectError
  yield actionChannel(() => {})
  // $ExpectError
  yield actionChannel(() => {}, actionBuffer)

  /* stringable action creator */

  yield actionChannel(stringableActionCreator)

  yield actionChannel(stringableActionCreator, buffers.fixed<MyAction>())
  // $ExpectError
  yield actionChannel(stringableActionCreator, nonActionBuffer)

  /* array */
```

```
  yield actionChannel(['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator])

  // $ExpectError
  yield actionChannel([() => {}])
}

function* testCancelled(): SagaIterator {
  yield cancelled()
  // $ExpectError
  yield cancelled(1)
}

function* testFlush(): SagaIterator {
  // $ExpectError
  yield flush()
  // $ExpectError
  yield flush({})

  yield flush(channel)
  yield flush(eventChannel)
  // $ExpectError
  yield flush(multicastChannel)
}

function* testGetContext(): SagaIterator {
  // $ExpectError
  yield getContext()

  // $ExpectError
  yield getContext({})

  yield getContext('prop')
}

function* testSetContext(): SagaIterator {
  // $ExpectError
  yield setContext()

  // $ExpectError
  yield setContext('prop')

  yield setContext({ prop: 1 })
}

function* testTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery()
  // $ExpectError
  yield takeEvery('my-action')

  yield takeEvery('my-action', (action: Action) => {})
  yield takeEvery('my-action', (action: MyAction) => {})
  yield takeEvery('my-action', function*(action: Action): SagaIterator {})
  yield takeEvery('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker1)
  // $ExpectError
  yield takeEvery('my-action', helperWorker1, 1)
  yield takeEvery('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga1)
  // $ExpectError
  yield takeEvery('my-action', helperSaga1, 1)
  yield takeEvery('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  const helperWorker8 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g') => {}

  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperWorker8, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeEvery('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeEvery(isMyAction, action => action.customField)

  yield takeEvery(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeEvery(() => {}, (action: Action) => {})

  yield takeEvery(stringableActionCreator, action => action.customField)
```

```
  yield takeEvery(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeEvery(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeEvery([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeEvery(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeEvery(): SagaIterator {
  // $ExpectError
  yield takeEvery(channel)

  // $ExpectError
  yield takeEvery(channel, (action: Action) => {})
  yield takeEvery(channel, (action: ChannelItem) => {})
  yield takeEvery(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeEvery(channel, helperWorker1)
  // $ExpectError
  yield takeEvery(channel, helperWorker1, 1)
  yield takeEvery(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga1)
  // $ExpectError
  yield takeEvery(channel, helperSaga1, 1)
  yield takeEvery(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeEvery(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeEvery(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeEvery(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeEvery(eventChannel, (action: ChannelItem) => {})
  yield takeEvery(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest()
  // $ExpectError
  yield takeLatest('my-action')

  yield takeLatest('my-action', (action: Action) => {})
  yield takeLatest('my-action', (action: MyAction) => {})
  yield takeLatest('my-action', function*(action: Action): SagaIterator {})
  yield takeLatest('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker1)
  // $ExpectError
  yield takeLatest('my-action', helperWorker1, 1)
  yield takeLatest('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
```

```
  yield takeLatest('my-action', helperSaga1)
  // $ExpectError
  yield takeLatest('my-action', helperSaga1, 1)
  yield takeLatest('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLatest('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLatest(isMyAction, action => action.customField)

  yield takeLatest(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLatest(() => {}, (action: Action) => {})

  yield takeLatest(stringableActionCreator, action => action.customField)

  yield takeLatest(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLatest(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLatest([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLatest(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLatest(): SagaIterator {
  // $ExpectError
  yield takeLatest(channel)

  // $ExpectError
  yield takeLatest(channel, (action: Action) => {})
  yield takeLatest(channel, (action: ChannelItem) => {})
  yield takeLatest(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLatest(channel, helperWorker1)
  // $ExpectError
  yield takeLatest(channel, helperWorker1, 1)
  yield takeLatest(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga1)
  // $ExpectError
  yield takeLatest(channel, helperSaga1, 1)
  yield takeLatest(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}
```

```
  // $ExpectError
  yield takeLatest(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLatest(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLatest(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLatest(eventChannel, (action: ChannelItem) => {})
  yield takeLatest(multicastChannel, (action: ChannelItem) => {})
}

function* testTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading()
  // $ExpectError
  yield takeLeading('my-action')

  yield takeLeading('my-action', (action: Action) => {})
  yield takeLeading('my-action', (action: MyAction) => {})
  yield takeLeading('my-action', function*(action: Action): SagaIterator {})
  yield takeLeading('my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker1)
  // $ExpectError
  yield takeLeading('my-action', helperWorker1, 1)
  yield takeLeading('my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga1)
  // $ExpectError
  yield takeLeading('my-action', helperSaga1, 1)
  yield takeLeading('my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield takeLeading('my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading((action: Action) => action.type === 'my-action', (action: Action) => {})
  yield takeLeading(isMyAction, action => action.customField)

  yield takeLeading(
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield takeLeading(() => {}, (action: Action) => {})

  yield takeLeading(stringableActionCreator, action => action.customField)

  yield takeLeading(
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield takeLeading(
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield takeLeading([pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield takeLeading(
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
```

```
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelTakeLeading(): SagaIterator {
  // $ExpectError
  yield takeLeading(channel)

  // $ExpectError
  yield takeLeading(channel, (action: Action) => {})
  yield takeLeading(channel, (action: ChannelItem) => {})
  yield takeLeading(channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield takeLeading(channel, helperWorker1)
  // $ExpectError
  yield takeLeading(channel, helperWorker1, 1)
  yield takeLeading(channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga1)
  // $ExpectError
  yield takeLeading(channel, helperSaga1, 1)
  yield takeLeading(channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield takeLeading(channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield takeLeading(channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield takeLeading(channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield takeLeading(eventChannel, (action: ChannelItem) => {})
  yield takeLeading(multicastChannel, (action: ChannelItem) => {})
}

function* testThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1)
  // $ExpectError
  yield throttle(1, 'my-action')

  yield throttle(1, 'my-action', (action: Action) => {})
  yield throttle(1, 'my-action', (action: MyAction) => {})
  yield throttle(1, 'my-action', function*(action: Action): SagaIterator {})
  yield throttle(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1)
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker1, 1)
  yield throttle(1, 'my-action', helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga1)
  // $ExpectError
  yield throttle(1, 'my-action', helperSaga1, 1)
  yield throttle(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield throttle(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield throttle(1, isMyAction, action => action.customField)

  yield throttle(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield throttle(1, () => {}, (action: Action) => {})

  yield throttle(1, stringableActionCreator, action => action.customField)
```

```
  yield throttle(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield throttle(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield throttle(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield throttle(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelThrottle(): SagaIterator {
  // $ExpectError
  yield throttle(1, channel)

  // $ExpectError
  yield throttle(1, channel, (action: Action) => {})
  yield throttle(1, channel, (action: ChannelItem) => {})
  yield throttle(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker1)
  // $ExpectError
  yield throttle(1, channel, helperWorker1, 1)
  yield throttle(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga1)
  // $ExpectError
  yield throttle(1, channel, helperSaga1, 1)
  yield throttle(1, channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield throttle(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield throttle(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield throttle(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield throttle(1, eventChannel, (action: ChannelItem) => {})
  yield throttle(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1)
  // $ExpectError
  yield debounce(1, 'my-action')

  yield debounce(1, 'my-action', (action: Action) => {})
  yield debounce(1, 'my-action', (action: MyAction) => {})
  yield debounce(1, 'my-action', function*(action: Action): SagaIterator {})
  yield debounce(1, 'my-action', function*(action: MyAction): SagaIterator {})

  const helperWorker1 = (a: 'a', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1)
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker1, 1)
  yield debounce(1, 'my-action', helperWorker1, 'a')
```

```
  function* helperSaga1(a: 'a', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1)
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga1, 1)
  yield debounce(1, 'my-action', helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction) => {}

  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: MyAction): SagaIterator {}

  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  // $ExpectError
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f')
  yield debounce(1, 'my-action', helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, (action: Action) => action.type === 'my-action', (action: Action) => {})
  yield debounce(1, isMyAction, action => action.customField)

  yield debounce(
    1,
    isMyAction,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  // $ExpectError
  yield debounce(1, () => {}, (action: Action) => {})

  yield debounce(1, stringableActionCreator, action => action.customField)

  yield debounce(
    1,
    stringableActionCreator,
    (a: { foo: string }, action: MyAction) => {
      a.foo + action.customField
    },
    { foo: 'bar' },
  )

  yield debounce(
    1,
    ['my-action', (action: Action) => action.type === 'my-action', stringableActionCreator, isMyAction],
    (action: Action) => {},
  )

  // test inference of action types from action pattern
  const pattern1: StringableActionCreator<{ type: 'A' }> = null!
  const pattern2: StringableActionCreator<{ type: 'B' }> = null!

  yield debounce(1, [pattern1, pattern2], action => {
    if (action.type === 'A') {
    }

    if (action.type === 'B') {
    }

    // $ExpectError
    if (action.type === 'C') {
    }
  })
  yield debounce(
    1,
    [pattern1, pattern2],
    (arg: { foo: string }, action: ActionMatchingPattern<typeof pattern1 | typeof pattern2>) => {
      if (action.type === 'A') {
      }

      if (action.type === 'B') {
      }

      // $ExpectError
      if (action.type === 'C') {
      }
    },
    { foo: 'bar' },
  )
}

function* testChannelDebounce(): SagaIterator {
  // $ExpectError
  yield debounce(1, channel)

  // $ExpectError
  yield debounce(1, channel, (action: Action) => {})
  yield debounce(1, channel, (action: ChannelItem) => {})
  yield debounce(1, channel, action => {
    // $ExpectError
    action.foo
    action.someField
  })

  const helperWorker1 = (a: 'a', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker1)
  // $ExpectError
  yield debounce(1, channel, helperWorker1, 1)
  yield debounce(1, channel, helperWorker1, 'a')

  function* helperSaga1(a: 'a', action: ChannelItem): SagaIterator {}
```

```
  // $ExpectError
  yield debounce(1, channel, helperSaga1)
  // $ExpectError
  yield debounce(1, channel, helperSaga1, 1)
  yield debounce(1, channel, helperSaga1, 'a')

  const helperWorker7 = (a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem) => {}

  // $ExpectError
  yield debounce(1, channel, helperWorker7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperWorker7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  function* helperSaga7(a: 'a', b: 'b', c: 'c', d: 'd', e: 'e', f: 'f', g: 'g', action: ChannelItem): SagaIterator {}

  // $ExpectError
  yield debounce(1, channel, helperSaga7, 1, 'b', 'c', 'd', 'e', 'f', 'g')
  yield debounce(1, channel, helperSaga7, 'a', 'b', 'c', 'd', 'e', 'f', 'g')

  yield debounce(1, eventChannel, (action: ChannelItem) => {})
  yield debounce(1, multicastChannel, (action: ChannelItem) => {})
}

function* testDelay(): SagaIterator {
  // $ExpectError
  yield delay()
  yield delay(1)
}

function* testRetry(): SagaIterator {
  // $ExpectError
  yield retry()
  // $ExpectError
  yield retry(1, 0, 1)
  yield retry(1, 0, () => 1)

  yield retry<() => 'foo'>(1, 0, () => 'foo')
  // $ExpectError
  yield retry<() => 'bar'>(1, 0, () => 'foo')

  yield retry(1, 0, a => a + 1, 42)
  // $ExpectError
  yield retry(1, 0, (a: string) => a, 42)
  // $ExpectError
  yield retry<(a: string) => number>(1, 0, a => a, 42)

  yield retry(1, 0, (a: number, b: number, c: string) => a, 1, 2, '3')
}

declare const promise: Promise<any>

function* testAll(): SagaIterator {
  yield all([call(() => {})])

  // $ExpectError
  yield all([1])

  // $ExpectError
  yield all([() => {}])

  // $ExpectError
  yield all([promise])

  // $ExpectError
  yield all([1, () => {}, promise])

  yield all({
    named: call(() => {}),
  })

  // $ExpectError
  yield all({
    named: 1,
  })

  // $ExpectError
  yield all({
    named: () => {},
  })

  // $ExpectError
  yield all({
    named: promise,
  })

  // $ExpectError
  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testNonStrictAll() {
  yield all([1])

  yield all([() => {}])

  yield all([promise])

  yield all([1, () => {}, promise])

  yield all({
    named: 1,
  })

  yield all({
    named: () => {},
  })

  yield all({
    named: promise,
  })
```

```
  yield all({
    named1: 1,
    named2: () => {},
    named3: promise,
  })
}

function* testRace(): SagaIterator {
  yield race({
    call: call(() => {}),
  })

  // $ExpectError
  yield race({
    named: 1,
  })

  // $ExpectError
  yield race({
    named: () => {},
  })

  // $ExpectError
  yield race({
    named: promise,
  })

  // $ExpectError
  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  // $ExpectError
  yield race([...effectArray, promise])
}

function* testNonStrictRace() {
  yield race({
    named: 1,
  })

  yield race({
    named: () => {},
  })

  yield race({
    named: promise,
  })

  yield race({
    named1: 1,
    named2: () => {},
    named3: promise,
  })

  const effectArray = [call(() => {}), call(() => {})]
  yield race([...effectArray])
  yield race([...effectArray, promise])
}
```

../redux-saga/packages/core/types/ts4.2/index.d.ts

```
// TypeScript Version: 4.2

import { Saga, Buffer, Channel, END as EndType, Predicate, SagaIterator, Task, NotUndefined } from '@redux-saga/types'
import { ForkEffect } from './effects'

export { Saga, SagaIterator, Buffer, Channel, Task }

export type Action<T extends string = string> = {
  type: T
}

export interface AnyAction extends Action {
  [extraProps: string]: any
}

export interface UnknownAction extends Action {
  [extraProps: string]: unknown
}

interface Dispatch<A extends Action = UnknownAction> {
  <T extends A>(action: T, ...extraArgs: any[]): T
}

interface MiddlewareAPI<D extends Dispatch = Dispatch, S = any> {
  dispatch: D
  getState(): S
}

export interface Middleware<_DispatchExt = {}, S = any, D extends Dispatch = Dispatch> {
  (api: MiddlewareAPI<D, S>): (next: (action: never) => unknown) => (action: unknown) => unknown
}

/**
 * Used by the middleware to dispatch monitoring events. Actually the middleware
 * dispatches 6 events:
 *
 * - When a root saga is started (via `runSaga` or `sagaMiddleware.run`) the
 *   middleware invokes `sagaMonitor.rootSagaStarted`
 *
 * - When an effect is triggered (via `yield someEffect`) the middleware invokes
 *   `sagaMonitor.effectTriggered`
 *
```

```
 * - If the effect is resolved with success the middleware invokes
 *   `sagaMonitor.effectResolved`
 *
 * - If the effect is rejected with an error the middleware invokes
 *   `sagaMonitor.effectRejected`
 *
 * - If the effect is cancelled the middleware invokes
 *   `sagaMonitor.effectCancelled`
 *
 * - Finally, the middleware invokes `sagaMonitor.actionDispatched` when a Redux
 *   action is dispatched.
 */
export interface SagaMonitor {
  /**
   * @param effectId Unique ID assigned to this root saga execution
   * @param saga The generator function that starts to run
   * @param args The arguments passed to the generator function
   */
  rootSagaStarted?(options: { effectId: number; saga: Saga; args: any[] }): void
  /**
   * @param effectId Unique ID assigned to the yielded effect
   * @param parentEffectId ID of the parent Effect. In the case of a `race` or
   *   `parallel` effect, all effects yielded inside will have the direct
   *   race/parallel effect as a parent. In case of a top-level effect, the
   *   parent will be the containing Saga
   * @param label In case of a `race`/`all` effect, all child effects will be
   *   assigned as label the corresponding keys of the object passed to
   *   `race`/`all`
   * @param effect The yielded effect itself
   */
  effectTriggered?(options: { effectId: number; parentEffectId: number; label?: string; effect: any }): void
  /**
   * @param effectId The ID of the yielded effect
   * @param result The result of the successful resolution of the effect. In
   *   case of `fork` or `spawn` effects, the result will be a `Task` object.
   */
  effectResolved?(effectId: number, result: any): void
  /**
   * @param effectId The ID of the yielded effect
   * @param error Error raised with the rejection of the effect
   */
  effectRejected?(effectId: number, error: any): void
  /**
   * @param effectId The ID of the yielded effect
   */
  effectCancelled?(effectId: number): void
  /**
   * @param action The dispatched Redux action. If the action was dispatched by
   * a Saga then the action will have a property `SAGA_ACTION` set to true
   * (`SAGA_ACTION` can be imported from `@redux-saga/symbols`).
   */
  actionDispatched?(action: Action): void
}

/**
 * Creates a Redux middleware and connects the Sagas to the Redux Store
 *
 * #### Example
 *
 * Below we will create a function `configureStore` which will enhance the Store
 * with a new method `runSaga`. Then in our main module, we will use the method
 * to start the root Saga of the application.
 *
 * **configureStore.js**
 *
 *    import createSagaMiddleware from 'redux-saga'
 *    import reducer from './path/to/reducer'
 *
 *    export default function configureStore(initialState) {
 *      // Note: passing middleware as the last argument to createStore requires redux@>=3.1.0
 *      const sagaMiddleware = createSagaMiddleware()
 *      return {
 *        ...createStore(reducer, initialState, applyMiddleware(... other middleware ..., sagaMiddleware)),
 *        runSaga: sagaMiddleware.run
 *      }
 *    }
 *
 * **main.js**
 *
 *    import configureStore from './configureStore'
 *    import rootSaga from './sagas'
 *    // ... other imports
 *
 *    const store = configureStore()
 *    store.runSaga(rootSaga)
 *
 * @param options A list of options to pass to the middleware
 */
export default function createSagaMiddleware<C extends object>(options?: SagaMiddlewareOptions<C>): SagaMiddleware<C>

export interface SagaMiddlewareOptions<C extends object = {}> {
  /**
   * Initial value of the saga's context.
   */
  context?: C
  /**
   * If a Saga Monitor is provided, the middleware will deliver monitoring
   * events to the monitor.
   */
  sagaMonitor?: SagaMonitor
  /**
   * If provided, the middleware will call it with uncaught errors from Sagas.
   * useful for sending uncaught exceptions to error tracking services.
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * Allows you to intercept any effect, resolve it on your own and pass to the
   * next middleware.
   */
  effectMiddlewares?: EffectMiddleware[]
  /**
   * If provided, the middleware will use this channel instead of the default `stdChannel` for
```

```
   * take and put effects.
   */
  channel?: MulticastChannel<Action>
}

export interface SagaMiddleware<C extends object = {}> extends Middleware {
  /**
   * Dynamically run `saga`. Can be used to run Sagas **only after** the
   * `applyMiddleware` phase.
   *
   * The method returns a `Task` descriptor.
   *
   * #### Notes
   *
   * `saga` must be a function which returns a [Generator
   * Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator).
   * The middleware will then iterate over the Generator and execute all yielded
   * Effects.
   *
   * `saga` may also start other sagas using the various Effects provided by the
   * library. The iteration process described below is also applied to all child
   * sagas.
   *
   * In the first iteration, the middleware invokes the `next()` method to
   * retrieve the next Effect. The middleware then executes the yielded Effect
   * as specified by the Effects API below. Meanwhile, the Generator will be
   * suspended until the effect execution terminates. Upon receiving the result
   * of the execution, the middleware calls `next(result)` on the Generator
   * passing it the retrieved result as an argument. This process is repeated
   * until the Generator terminates normally or by throwing some error.
   *
   * If the execution results in an error (as specified by each Effect creator)
   * then the `throw(error)` method of the Generator is called instead. If the
   * Generator function defines a `try/catch` surrounding the current yield
   * instruction, then the `catch` block will be invoked by the underlying
   * Generator runtime. The runtime will also invoke any corresponding finally
   * block.
   *
   * In the case a Saga is cancelled (either manually or using the provided
   * Effects), the middleware will invoke `return()` method of the Generator.
   * This will cause the Generator to skip directly to the finally block.
   *
   * @param saga a Generator function
   * @param args arguments to be provided to `saga`
   */
  run<S extends Saga>(saga: S, ...args: Parameters<S>): Task

  setContext(props: Partial<C>): void
}

export interface EffectMiddleware {
  (next: (effect: any) => void): (effect: any) => void
}

/**
 * Allows starting sagas outside the Redux middleware environment. Useful if you
 * want to connect a Saga to external input/output, other than store actions.
 *
 * `runSaga` returns a Task object. Just like the one returned from a `fork`
 * effect.
 */
export function runSaga<Action, State, S extends Saga>(
  options: RunSagaOptions<Action, State>,
  saga: S,
  ...args: Parameters<S>
): Task

interface ErrorInfo {
  sagaStack: string
}

/**
 * The `{subscribe, dispatch}` is used to fulfill `take` and `put` Effects. This
 * defines the Input/Output interface of the Saga.
 *
 * `subscribe` is used to fulfill `take(PATTERN)` effects. It must call
 * `callback` every time it has an input to dispatch (e.g. on every mouse click
 * if the Saga is connected to DOM click events). Each time `subscribe` emits an
 * input to its callbacks, if the Saga is blocked on a `take` effect, and if the
 * take pattern matches the currently incoming input, the Saga is resumed with
 * that input.
 *
 * `dispatch` is used to fulfill `put` effects. Each time the Saga emits a
 * `yield put(output)`, `dispatch` is invoked with output.
 */
export interface RunSagaOptions<A, S> {
  /**
   * See docs for `channel`
   */
  channel?: PredicateTakeableChannel<A>
  /**
   * Used to fulfill `put` effects.
   *
   * @param output argument provided by the Saga to the `put` Effect
   */
  dispatch?(output: A): any
  /**
   * Used to fulfill `select` and `getState` effects
   */
  getState?(): S
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  sagaMonitor?: SagaMonitor
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  onError?(error: Error, errorInfo: ErrorInfo): void
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  context?: object
```

```
  /**
   * See docs for `createSagaMiddleware(options)`
   */
  effectMiddlewares?: EffectMiddleware[]
}

export const CANCEL: string
export const END: EndType
export type END = EndType

export interface TakeableChannel<T> {
  take(cb: (message: T | END) => void): void
}

export interface PuttableChannel<T> {
  put(message: T | END): void
}

export interface FlushableChannel<T> {
  flush(cb: (items: T[] | END) => void): void
}

/**
 * A factory method that can be used to create Channels. You can optionally pass
 * it a buffer to control how the channel buffers the messages.
 *
 * By default, if no buffer is provided, the channel will queue incoming
 * messages up to 10 until interested takers are registered. The default
 * buffering will deliver message using a FIFO strategy: a new taker will be
 * delivered the oldest message in the buffer.
 */
export function channel<T extends NotUndefined>(buffer?: Buffer<T>): Channel<T>

/**
 * Creates channel that will subscribe to an event source using the `subscribe`
 * method. Incoming events from the event source will be queued in the channel
 * until interested takers are registered.
 *
 * To notify the channel that the event source has terminated, you can notify
 * the provided subscriber with an `END`
 *
 * #### Example
 *
 * In the following example we create an event channel that will subscribe to a
 * `setInterval`
 *
 *    const countdown = (secs) => {
 *      return eventChannel(emitter => {
 *          const iv = setInterval(() => {
 *            console.log('countdown', secs)
 *            secs -= 1
 *            if (secs > 0) {
 *              emitter(secs)
 *            } else {
 *              emitter(END)
 *              clearInterval(iv)
 *              console.log('countdown terminated')
 *            }
 *          }, 1000);
 *          return () => {
 *            clearInterval(iv)
 *            console.log('countdown cancelled')
 *          }
 *        }
 *      )
 *    }
 *
 * @param subscribe used to subscribe to the underlying event source. The
 *   function must return an unsubscribe function to terminate the subscription.
 * @param buffer optional Buffer object to buffer messages on this channel. If
 *   not provided, messages will not be buffered on this channel.
 */
export function eventChannel<T extends NotUndefined>(subscribe: Subscribe<T>, buffer?: Buffer<T>): EventChannel<T>

export type Subscribe<T> = (cb: (input: T | END) => void) => Unsubscribe
export type Unsubscribe = () => void

export interface EventChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void): void
  flush(cb: (items: T[] | END) => void): void
  close(): void
}

export interface PredicateTakeableChannel<T> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
}

export interface MulticastChannel<T extends NotUndefined> {
  take(cb: (message: T | END) => void, matcher?: Predicate<T>): void
  put(message: T | END): void
  close(): void
}

export function multicastChannel<T extends NotUndefined>(): MulticastChannel<T>
export function stdChannel<T extends NotUndefined>(): MulticastChannel<T>

export function detach(forkEffect: ForkEffect): ForkEffect

/**
 * Provides some common buffers
 */
export const buffers: {
  /**
   * No buffering, new messages will be lost if there are no pending takers
   */
  none<T>(): Buffer<T>
  /**
   * New messages will be buffered up to `limit`. Overflow will raise an Error.
   * Omitting a `limit` value will result in a limit of 10.
   */
  fixed<T>(limit?: number): Buffer<T>
  /**
```

```
   * Like `fixed` but Overflow will cause the buffer to expand dynamically.
   */
  expanding<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will silently drop the messages.
   */
  dropping<T>(limit?: number): Buffer<T>
  /**
   * Same as `fixed` but Overflow will insert the new message at the end and
   * drop the oldest message in the buffer.
   */
  sliding<T>(limit?: number): Buffer<T>
}
```

../redux-saga/packages/core/types/ts4.2/middleware.test.ts

```
import createSagaMiddleware, { SagaIterator } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'
import { applyMiddleware } from 'redux'

function testApplyMiddleware() {
  const middleware = createSagaMiddleware()

  const enhancer = applyMiddleware(middleware)
}

declare const effect: StrictEffect
declare const promise: Promise<any>

function testRun() {
  const middleware = createSagaMiddleware()

  middleware.run(function* saga(): SagaIterator {})

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // middleware.run(function* saga(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a'): SagaIterator {}, 1)

  middleware.run(function* saga(a: 'a'): SagaIterator {}, 'a')

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
  }

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

  // $ExpectError
  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

  middleware.run(function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

  // test with any iterator i.e. when generator doesn't always yield Effects.
  middleware.run(function* saga() {
    yield promise
  })
}

function testOptions() {
  const emptyOptions = createSagaMiddleware({})

  const withOptions = createSagaMiddleware({
    onError(error) {
      console.error(error)
    },

    sagaMonitor: {
      effectTriggered() {},
    },

    effectMiddlewares: [
      next => effect => {
        setTimeout(() => {
          next(effect)
        }, 10)
      },
      next => effect => {
        setTimeout(() => {
          next(effect)
        }, 10)
      },
    ],
  })

  const withMonitor = createSagaMiddleware({
    sagaMonitor: {
      effectTriggered() {},
      effectResolved() {},
      effectRejected() {},
      effectCancelled() {},
      actionDispatched() {},
    },
  })
}

function testContext() {
  interface Context {
    a: string
    b: number
  }

  // $ExpectError
```

```
  createSagaMiddleware<Context>({ context: { c: 42 } })

  // $ExpectError
  createSagaMiddleware({ context: 42 })

  const middleware = createSagaMiddleware<Context>({
    context: { a: '', b: 42 },
  })

  // $ExpectError
  middleware.setContext({ c: 42 })

  middleware.setContext({ b: 42 })

  const task = middleware.run(function*() {
    yield effect
  })
  task.setContext({ b: 42 })

  task.setContext<Context>({ a: '' })
  // $ExpectError
  task.setContext<Context>({ c: '' })
}
```

../redux-saga/packages/core/types/ts4.2/runSaga.test.ts

```
import { SagaIterator, Task, runSaga, END, MulticastChannel } from 'redux-saga'
import { StrictEffect } from 'redux-saga/effects'

declare const stdChannel: MulticastChannel<any>
declare const promise: Promise<any>
declare const effect: StrictEffect
declare const iterator: Iterator<any>

function testRunSaga() {
  const task0: Task = runSaga<{ foo: string }, { baz: boolean }, () => SagaIterator>(
    {
      context: { a: 42 },

      channel: stdChannel,

      effectMiddlewares: [
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
        next => effect => {
          setTimeout(() => {
            next(effect)
          }, 10)
        },
      ],

      getState() {
        return { baz: true }
      },

      dispatch(input) {
        input.foo
        // $ExpectError
        input.bar
      },

      sagaMonitor: {
        effectTriggered() {},
        effectResolved() {},
        effectRejected() {},
        effectCancelled() {},
        actionDispatched() {},
      },

      onError(error) {
        console.error(error)
      },
    },
    function* saga(): SagaIterator {
      yield effect
    },
  )

  // $ExpectError
  runSaga()

  // $ExpectError
  runSaga({})

  // $ExpectError
  runSaga({}, iterator)

  runSaga({}, function* saga() {
    yield effect
  })

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // runSaga({}, function* saga(a: 'a'): SagaIterator {})
  }

  // $ExpectError
  runSaga({}, function* saga(a: 'a'): SagaIterator {}, 1)

  runSaga({}, function* saga(a: 'a'): SagaIterator {}, 'a')

  // TODO: https://github.com/Microsoft/TypeScript/issues/28803
  {
    // // $ExpectError
    // runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a')
```

```
  }

  // $ExpectError
  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 1)

  // $ExpectError
  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 1, 'b')

  runSaga({}, function* saga(a: 'a', b: 'b'): SagaIterator {}, 'a', 'b')

  // test with any iterator i.e. when generator doesn't always yield Effects.
  runSaga({}, function* saga() {
    yield promise
  })

  // $ExpectError
  runSaga({ context: 42 }, function* saga(): SagaIterator {})
}
```

../redux-saga/packages/deferred/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/deferred/index.d.ts

```
export interface Deferred<R> {
  resolve(result: R): void
  reject(error: any): void
  promise: Promise<R>
}

export default function deferred<R>(): Deferred<R>

export function arrayOfDeferred<R>(length: number): Deferred<R>[]
```

../redux-saga/packages/deferred/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
    },
  }),
]
```

../redux-saga/packages/deferred/src/index.js

```
export default function deferred() {
  const def = {}
  def.promise = new Promise((resolve, reject) => {
    def.resolve = resolve
    def.reject = reject
  })
  return def
}

export function arrayOfDeferred(length) {
  const arr = []

  for (let i = 0; i < length; i++) {
    arr.push(deferred())
  }

  return arr
}
```

../redux-saga/packages/delay-p/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/delay-p/index.d.ts

```
export default function delayP<T = true>(ms: number, val?: T): Promise<T>
```

../redux-saga/packages/delay-p/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
    },
  }),
]
```

../redux-saga/packages/delay-p/src/index.js

```
import { CANCEL } from '@redux-saga/symbols'

const MAX_SIGNED_INT = 2147483647

export default function delayP(ms, val = true) {
  // https://developer.mozilla.org/en-US/docs/Web/API/setTimeout#maximum_delay_value
  if (process.env.NODE_ENV !== 'production' && ms > MAX_SIGNED_INT) {
    throw new Error('delay only supports a maximum value of ' + MAX_SIGNED_INT + 'ms')
  }
  let timeoutId
  const promise = new Promise((resolve) => {
    timeoutId = setTimeout(resolve, Math.min(MAX_SIGNED_INT, ms), val)
  })

  promise[CANCEL] = () => {
    clearTimeout(timeoutId)
  }

  return promise
}
```

../redux-saga/packages/is/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/is/index.d.ts

```
import { Action } from 'redux'
import { ActionPattern, Buffer, Channel, GuardPredicate, Pattern, Task, Effect } from '@redux-saga/types'

export const array: GuardPredicate<Array<any>>
export const buffer: GuardPredicate<Buffer<any>>
export const channel: GuardPredicate<Channel<any>>
export const effect: GuardPredicate<Effect>
export const func: GuardPredicate<Function>
export const iterable: GuardPredicate<Iterable<any>>
export const iterator: GuardPredicate<Iterator<any>>
export const notUndef: GuardPredicate<any>
export const number: GuardPredicate<number>
export const object: GuardPredicate<object>
export const observable: GuardPredicate<{ subscribe: Function }>
export const pattern: GuardPredicate<Pattern<any> | ActionPattern>
export const promise: GuardPredicate<Promise<any>>
export const string: GuardPredicate<string>
export const stringableFunc: GuardPredicate<Function>
export const task: GuardPredicate<Task>
export const sagaAction: GuardPredicate<Action & { '@@redux-saga/SAGA_ACTION': true }>
export const undef: GuardPredicate<undefined>
```

../redux-saga/packages/is/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
```

```
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
    },
  }),
]
```

../redux-saga/packages/is/src/index.js

```
import { TASK, MULTICAST, IO, SAGA_ACTION } from '@redux-saga/symbols'

export const undef = (v) => v === null || v === undefined
export const notUndef = (v) => v !== null && v !== undefined
export const func = (f) => typeof f === 'function'
export const number = (n) => typeof n === 'number'
export const string = (s) => typeof s === 'string'
export const array = Array.isArray
export const object = (obj) => obj && !array(obj) && typeof obj === 'object'
export const promise = (p) => p && func(p.then)
export const iterator = (it) => it && func(it.next) && func(it.throw)
export const iterable = (it) => (it && func(Symbol) ? func(it[Symbol.iterator]) : array(it))
export const task = (t) => t && t[TASK]
export const sagaAction = (a) => Boolean(a && a[SAGA_ACTION])
export const observable = (ob) => ob && func(ob.subscribe)
export const buffer = (buf) => buf && func(buf.isEmpty) && func(buf.take) && func(buf.put)
export const pattern = (pat) => pat && (string(pat) || symbol(pat) || func(pat) || (array(pat) && pat.every(pattern)))
export const channel = (ch) => ch && func(ch.take) && func(ch.close)
export const stringableFunc = (f) => func(f) && f.hasOwnProperty('toString')
export const symbol = (sym) =>
  Boolean(sym) && typeof Symbol === 'function' && sym.constructor === Symbol && sym !== Symbol.prototype
export const multicast = (ch) => channel(ch) && ch[MULTICAST]
export const effect = (eff) => eff && eff[IO]
```

../redux-saga/packages/redux-saga/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
        exclude: ['transform-regenerator'],
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/redux-saga/effects.d.ts

```
export * from '@redux-saga/core/effects'
```

../redux-saga/packages/redux-saga/index.d.ts

```
export * from '@redux-saga/core'
export { default } from '@redux-saga/core'
```

../redux-saga/packages/redux-saga/rollup.config.js

```
import * as path from 'path'
import alias from 'rollup-plugin-alias'
import nodeResolve from 'rollup-plugin-node-resolve'
import babel from 'rollup-plugin-babel'
import replace from 'rollup-plugin-replace'
import { terser } from 'rollup-plugin-terser'
import { rollup as lernaAlias } from 'lerna-alias'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

let aliases = lernaAlias()
aliases = {
  '@redux-saga/core/effects': aliases['@redux-saga/core'].replace(/index\.js$/, 'effects.js'),
  '@babel/runtime/helpers/extends': require.resolve('@babel/runtime/helpers/esm/extends'),
  ...aliases,
}

const presetEnvPath = require.resolve('@babel/preset-env')
```

```
const createConfig = ({
  input,
  output,
  external,
  env,
  min = false,
  useESModules = output.format !== 'cjs',
  esmodulesBrowsersTarget = false,
}) => ({
  input,
  output: {
    name: 'ReduxSaga',
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(external === 'peers' ? peerDeps : deps.concat(peerDeps)),
  treeshake: {
    propertyReadSideEffects: false,
  },
  plugins: [
    alias(aliases),
    nodeResolve({
      jsnext: true,
    }),
    babel.custom(() => {
      if (!esmodulesBrowsersTarget) {
        return {}
      }
      return {
        config(config) {
          return {
            ...config.options,
            presets: config.options.presets.map((preset) => {
              if (preset.file.resolved !== presetEnvPath) {
                return preset
              }

              return [
                presetEnvPath,
                {
                  ...preset.options,
                  targets: { esmodules: true },
                },
              ]
            }),
          }
        },
      }
    })({
      exclude: 'node_modules/**',
      babelrcRoots: path.resolve(__dirname, '../*'),
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
    env &&
      replace({
        'process.env.NODE_ENV': JSON.stringify(env),
      }),
    min &&
      terser({
        compress: {
          pure_getters: true,
          unsafe: true,
          unsafe_comps: true,
          warnings: false,
        },
      }),
  ].filter(Boolean),
  onwarn(warning, warn) {
    if (warning.code === 'UNUSED_EXTERNAL_IMPORT') {
      return
    }
    warn(warning)
  },
})

const multiInput = {
  core: 'src/index.js',
  effects: 'src/effects.js',
}

const developmentBase = {
  external: 'peers',
  env: 'development',
}

const productionBase = {
  external: 'peers',
  env: 'production',
  min: true,
}

export default [
  ...['esm', 'cjs'].map((format) =>
    createConfig({
      input: multiInput,
      output: {
        dir: 'dist',
        format,
        entryFileNames: 'redux-saga-[name]-npm-proxy.[format].js',
      },
    }),
  ),
  createConfig({
```

```
    ...developmentBase,
    input: 'src/index.umd.js',
    output: {
      file: pkg.unpkg.replace(/\.min\.js$/, '.js'),
      format: 'umd',
    },
  }),
  createConfig({
    ...productionBase,
    input: 'src/index.umd.js',
    output: {
      file: pkg.unpkg,
      format: 'umd',
    },
  }),
  createConfig({
    ...developmentBase,
    input: 'src/effects.js',
    output: {
      file: 'dist/redux-saga-effects.umd.js',
      format: 'umd',
      name: 'ReduxSagaEffects',
    },
  }),
  createConfig({
    ...productionBase,
    input: 'src/effects.js',
    output: {
      file: 'dist/redux-saga-effects.umd.min.js',
      format: 'umd',
      name: 'ReduxSagaEffects',
    },
  }),
  createConfig({
    ...developmentBase,
    input: multiInput,
    output: {
      dir: 'dist',
      format: 'esm',
      entryFileNames: 'redux-saga-[name].esmodules-browsers.js',
    },
    esmodulesBrowsersTarget: true,
  }),
  createConfig({
    ...productionBase,
    input: multiInput,
    output: {
      dir: 'dist',
      format: 'esm',
      entryFileNames: 'redux-saga-[name].esmodules-browsers.min.js',
    },
    esmodulesBrowsersTarget: true,
  }),
]
```

../redux-saga/packages/redux-saga/src/effects.js

```
export * from '@redux-saga/core/effects'
```

../redux-saga/packages/redux-saga/src/index.js

```
export * from '@redux-saga/core'
import createSagaMiddleware from '@redux-saga/core'
export default createSagaMiddleware
```

../redux-saga/packages/redux-saga/src/index.umd.js

```
export { default } from '.'
export * from '.'

import * as effects from './effects'
export { effects }
```

../redux-saga/packages/simple-saga-monitor/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/simple-saga-monitor/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
```

```
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
    },
  }),
]
```

../redux-saga/packages/simple-saga-monitor/src/index.js

```
/* eslint-disable no-console */
import * as is from '@redux-saga/is'
import { CANCELLED, IS_BROWSER, PENDING, IS_REACT_NATIVE, REJECTED, RESOLVED } from './modules/constants'
import { isRaceEffect } from './modules/checkers'
import logSaga from './modules/logSaga'
import Manager from './modules/Manager'

const globalScope = IS_REACT_NATIVE ? global : IS_BROWSER ? window : null

// `VERBOSE` can be made a setting configured from the outside.
const VERBOSE = false

function time() {
  if (typeof performance !== 'undefined' && performance.now) {
    return performance.now()
  } else {
    return Date.now()
  }
}

const manager = new Manager()

function rootSagaStarted(desc) {
  if (VERBOSE) {
    console.log('Root saga started:', desc.saga.name || 'anonymous', desc.args)
  }
  manager.setRootEffect(
    desc.effectId,
    Object.assign({}, desc, {
      status: PENDING,
      start: time(),
    }),
  )
}

function effectTriggered(desc) {
  if (VERBOSE) {
    console.log('Saga monitor: effectTriggered:', desc)
  }
  manager.set(
    desc.effectId,
    Object.assign({}, desc, {
      status: PENDING,
      start: time(),
    }),
  )
}

function effectResolved(effectId, result) {
  if (VERBOSE) {
    console.log('Saga monitor: effectResolved:', effectId, result)
  }
  resolveEffect(effectId, result)
}

function effectRejected(effectId, error) {
  if (VERBOSE) {
    console.log('Saga monitor: effectRejected:', effectId, error)
  }
  rejectEffect(effectId, error)
```

```
}

function effectCancelled(effectId) {
  if (VERBOSE) {
    console.log('Saga monitor: effectCancelled:', effectId)
  }
  cancelEffect(effectId)
}

function computeEffectDur(effect) {
  const now = time()
  Object.assign(effect, {
    end: now,
    duration: now - effect.start,
  })
}

function resolveEffect(effectId, result) {
  const effect = manager.get(effectId)

  if (is.task(result)) {
    result.toPromise().then(
      (taskResult) => {
        if (result.isCancelled()) {
          cancelEffect(effectId)
        } else {
          resolveEffect(effectId, taskResult)
        }
      },
      (taskError) => rejectEffect(effectId, taskError),
    )
  } else {
    computeEffectDur(effect)
    effect.status = RESOLVED
    effect.result = result
    if (isRaceEffect(effect.effect)) {
      setRaceWinner(effectId, result)
    }
  }
}

function rejectEffect(effectId, error) {
  const effect = manager.get(effectId)
  computeEffectDur(effect)
  effect.status = REJECTED
  effect.error = error
  if (isRaceEffect(effect.effect)) {
    setRaceWinner(effectId, error)
  }
}

function cancelEffect(effectId) {
  const effect = manager.get(effectId)
  computeEffectDur(effect)
  effect.status = CANCELLED
}

function setRaceWinner(raceEffectId, result) {
  const winnerLabel = Object.keys(result)[0]
  for (const childId of manager.getChildIds(raceEffectId)) {
    const childEffect = manager.get(childId)
    if (childEffect.label === winnerLabel) {
      childEffect.winner = true
    }
  }
}

// Export the snapshot-logging function to run from the browser console or extensions.
if (globalScope) {
  console.log('Enter `$$LogSagas()` to print the monitor log')
  globalScope.$$LogSagas = () => logSaga(manager)
}

// Export the snapshot-logging function for arbitrary use by external code.
export { logSaga }

// Export the `sagaMonitor` to pass to the middleware.
export default {
  rootSagaStarted,
  effectTriggered,
  effectResolved,
  effectRejected,
  effectCancelled,
  actionDispatched: () => {},
}
```

../redux-saga/packages/simple-saga-monitor/src/modules/DescriptorFormatter.js

```
import * as is from '@redux-saga/is'
import Formatter from './Formatter'
import { CANCELLED, PENDING, REJECTED, RESOLVED } from './constants'

const DEFAULT_STYLE = 'color: black'
const LABEL_STYLE = 'font-weight: bold'
const EFFECT_TYPE_STYLE = 'color: blue'
const ERROR_STYLE = 'color: red'
const CANCEL_STYLE = 'color: #ccc'

export default class DescriptorFormatter extends Formatter {
  constructor(isCancel, isError) {
    super()
    this.logMethod = isError ? 'error' : 'log'
    this.styleOverride = (s) => (isCancel ? CANCEL_STYLE : isError ? ERROR_STYLE : s)
  }

  resetStyle() {
    return this.add('%c', this.styleOverride(DEFAULT_STYLE))
  }
```

```
  addLabel(text) {
    if (text) {
      return this.add(`%c ${text} `, this.styleOverride(LABEL_STYLE))
    } else {
      return this
    }
  }

  addEffectType(text) {
    return this.add(`%c ${text} `, this.styleOverride(EFFECT_TYPE_STYLE))
  }

  addDescResult(descriptor, ignoreResult) {
    const { status, result, error, duration } = descriptor
    if (status === RESOLVED && !ignoreResult) {
      if (is.array(result)) {
        this.addValue(' ➔ ')
        this.addValue(result)
      } else {
        this.appendData('➔', result)
      }
    } else if (status === REJECTED) {
      this.appendData('➔ ⚠', error)
    } else if (status === PENDING) {
      this.appendData('⌛')
    } else if (status === CANCELLED) {
      this.appendData('➔ Cancelled!')
    }
    if (status !== PENDING) {
      this.appendData(`(${duration.toFixed(2)}ms)`)
    }

    return this
  }
}
```

../redux-saga/packages/simple-saga-monitor/src/modules/Formatter.js

```
import { IS_BROWSER } from './constants'

function argToString(arg) {
  return typeof arg === 'function' ? `${arg.name}` : typeof arg === 'string' ? `'${arg}'` : arg
}

function isPrimitive(val) {
  return (
    typeof val === 'string' ||
    typeof val === 'number' ||
    typeof val === 'boolean' ||
    typeof val === 'symbol' ||
    val === null ||
    val === undefined
  )
}

export default class Formatter {
  constructor() {
    this.logs = []
    this.suffix = []
  }

  add(msg, ...args) {
    // Remove the `%c` CSS styling that is not supported by the Node console.
    if (!IS_BROWSER && typeof msg === 'string') {
      const prevMsg = msg
      msg = msg.replace(/^%c\s*/, '')
      if (msg !== prevMsg) {
        // Remove the first argument which is the CSS style string.
        args.shift()
      }
    }
    this.logs.push({ msg, args })
    return this
  }

  appendData(...data) {
    this.suffix.push(...data)
    return this
  }

  addValue(value) {
    if (isPrimitive(value)) {
      this.add(value)
    } else {
      // The browser console supports `%O`, the Node console does not.
      if (IS_BROWSER) {
        this.add('%O', value)
      } else {
        this.add('%s', require('util').inspect(value))
      }
    }
    return this
  }

  addCall(name, args) {
    if (!args.length) {
      this.add(`${name}()`)
    } else {
      this.add(name)
      this.add('(')
      args.forEach((arg, i) => {
        this.addValue(argToString(arg))
        this.addValue(i === args.length - 1 ? ')' : ', ')
      })
    }
    return this
  }

  getLog() {
```

```
    const msgs = []
    const msgsArgs = []
    for (const { msg, args } of this.logs) {
      msgs.push(msg)
      msgsArgs.push(...args)
    }
    return [msgs.join(''), ...msgsArgs, ...this.suffix]
  }
}
```

../redux-saga/packages/simple-saga-monitor/src/modules/Manager.js

```
/** The manager is used for bookkeeping all the effect descriptors */
export default class Manager {
  constructor() {
    this.rootIds = []
    // effect-id-to-effect-descriptor
    this.map = {}
    // effect-id-to-array-of-child-id
    this.childIdsMap = {}
  }

  get(effectId) {
    return this.map[effectId]
  }

  set(effectId, desc) {
    this.map[effectId] = desc

    if (this.childIdsMap[desc.parentEffectId] == null) {
      this.childIdsMap[desc.parentEffectId] = []
    }
    this.childIdsMap[desc.parentEffectId].push(effectId)
  }

  setRootEffect(effectId, desc) {
    this.rootIds.push(effectId)
    this.set(effectId, Object.assign({ root: true }, desc))
  }

  getRootIds() {
    return this.rootIds
  }

  getChildIds(parentEffectId) {
    return this.childIdsMap[parentEffectId] || []
  }
}
```

../redux-saga/packages/simple-saga-monitor/src/modules/checkers.js

```
import * as is from '@redux-saga/is'
import { effectTypes } from 'redux-saga/effects'

export const isRaceEffect = (eff) => is.effect(eff) && eff.type === effectTypes.RACE
```

../redux-saga/packages/simple-saga-monitor/src/modules/consoleGroup.js

```
/* eslint-disable no-console */

// Poor man's `console.group` and `console.groupEnd` for Node.
// Can be overridden by the `console-group` polyfill.
// The poor man's groups look nice, too, so whether to use
// the polyfilled methods or the hand-made ones can be made a preference.
let groupPrefix = ''
const GROUP_SHIFT = '   '
const GROUP_ARROW = '▼'

export function consoleGroup(...args) {
  if (console.group) {
    console.group(...args)
  } else {
    console.log('')
    console.log(groupPrefix + GROUP_ARROW, ...args)
    groupPrefix += GROUP_SHIFT
  }
}

export function consoleGroupEnd() {
  if (console.groupEnd) {
    console.groupEnd()
  } else {
    groupPrefix = groupPrefix.substr(0, groupPrefix.length - GROUP_SHIFT.length)
  }
}
```

../redux-saga/packages/simple-saga-monitor/src/modules/constants.js

```
export const PENDING = 'PENDING'
export const RESOLVED = 'RESOLVED'
export const REJECTED = 'REJECTED'
export const CANCELLED = 'CANCELLED'

export const IS_BROWSER = typeof window !== 'undefined' && window.document
export const IS_REACT_NATIVE = typeof navigator !== 'undefined' && navigator.product === 'ReactNative'
```

../redux-saga/packages/simple-saga-monitor/src/modules/logSaga.js

```
/* eslint-disable no-console */
import * as is from '@redux-saga/is'
```

```
import { effectTypes } from 'redux-saga/effects'
import { consoleGroup, consoleGroupEnd } from './consoleGroup'
import { CANCELLED, REJECTED } from './constants'
import DescriptorFormatter from './DescriptorFormatter'

export default function logSaga(manager) {
  if (manager.getRootIds().length === 0) {
    console.log('Saga monitor: No effects to log')
  }
  console.log('')
  console.log('Saga monitor:', Date.now(), new Date().toISOString())
  for (const id of manager.getRootIds()) {
    logEffectTree(manager, id)
  }
  console.log('')
}

function logEffectTree(manager, effectId) {
  const desc = manager.get(effectId)
  const childIds = manager.getChildIds(effectId)

  const formatter = getFormatterFromDescriptor(desc)
  if (childIds.length === 0) {
    console[formatter.logMethod](...formatter.getLog())
  } else {
    consoleGroup(...formatter.getLog())
    for (const id of childIds) {
      logEffectTree(manager, id)
    }
    consoleGroupEnd()
  }
}

function getFormatterFromDescriptor(desc) {
  const isCancel = desc.status === CANCELLED
  const isError = desc.status === REJECTED

  const formatter = new DescriptorFormatter(isCancel, isError)

  const winnerInd = desc.winner ? (isError ? '✘' : '✓') : ''
  formatter.addLabel(winnerInd).addLabel(desc.label)

  if (desc.root) {
    formatter.addEffectType('root').resetStyle().addCall(desc.saga.name, desc.args).addDescResult(desc)
  } else if (is.iterator(desc.effect)) {
    formatter.addValue(desc.effect.name).addDescResult(desc, true)
  } else if (is.promise(desc.effect)) {
    formatter.addEffectType('promise').resetStyle().addDescResult(desc)
  } else if (is.effect(desc.effect)) {
    const { type, payload } = desc.effect

    if (type === effectTypes.TAKE) {
      formatter
        .addEffectType('take')
        .resetStyle()
        .addValue(payload.channel == null ? payload.pattern : payload)
        .addDescResult(desc)
    } else if (type === effectTypes.PUT) {
      formatter
        .addEffectType('put')
        .resetStyle()
        .addDescResult(Object.assign({}, desc, { result: payload }))
    } else if (type === effectTypes.ALL) {
      formatter.addEffectType('all').resetStyle().addDescResult(desc, true)
    } else if (type === effectTypes.RACE) {
      formatter.addEffectType('race').resetStyle().addDescResult(desc, true)
    } else if (type === effectTypes.CALL) {
      formatter.addEffectType('call').resetStyle().addCall(payload.fn.name, payload.args).addDescResult(desc)
    } else if (type === effectTypes.CPS) {
      formatter.addEffectType('cps').resetStyle().addCall(payload.fn.name, payload.args).addDescResult(desc)
    } else if (type === effectTypes.FORK) {
      formatter
        .addEffectType(payload.detached ? 'spawn' : 'fork')
        .resetStyle()
        .addCall(payload.fn.name, payload.args)
        .addDescResult(desc)
    } else if (type === effectTypes.JOIN) {
      formatter.addEffectType('join').resetStyle().addDescResult(desc)
    } else if (type === effectTypes.CANCEL) {
      formatter.addEffectType('cancel').resetStyle().appendData(payload.name)
    } else if (type === effectTypes.SELECT) {
      formatter.addEffectType('select').resetStyle().addCall(payload.selector.name, payload.args).addDescResult(desc)
    } else if (type === effectTypes.ACTION_CHANNEL) {
      formatter
        .addEffectType('actionChannel')
        .resetStyle()
        .addValue(payload.buffer == null ? payload.pattern : payload)
        .addDescResult(desc)
    } else if (type === effectTypes.CANCELLED) {
      formatter.addEffectType('cancelled').resetStyle().addDescResult(desc)
    } else if (type === effectTypes.FLUSH) {
      formatter.addEffectType('flush').resetStyle().addValue(payload).addDescResult(desc)
    } else if (type === effectTypes.GET_CONTEXT) {
      formatter.addEffectType('getContext').resetStyle().addValue(payload).addDescResult(desc)
    } else if (type === effectTypes.SET_CONTEXT) {
      formatter.addEffectType('setContext').resetStyle().addValue(payload).addDescResult(desc, true)
    } else {
      throw new Error(`Invalid effect type ${type}`)
    }
  } else {
    formatter.addEffectType('unknown').resetStyle().addDescResult(desc)
  }

  return formatter
}
```

../redux-saga/packages/symbols/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/symbols/index.d.ts

```
export const CANCEL: string
export const CHANNEL_END_TYPE: string
export const IO: string
export const MATCH: string
export const MULTICAST: string
export const SAGA_ACTION: string
export const SAGA_LOCATION: string
export const SELF_CANCELLATION: string
export const TASK: string
export const TASK_CANCEL: string
export const TERMINATE: string
```

../redux-saga/packages/symbols/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
    },
  }),
]
```

../redux-saga/packages/symbols/src/index.js

```
const createSymbol = (name) => `@@redux-saga/${name}`

export const CANCEL = createSymbol('CANCEL_PROMISE')
export const CHANNEL_END_TYPE = createSymbol('CHANNEL_END')
export const IO = createSymbol('IO')
export const MATCH = createSymbol('MATCH')
export const MULTICAST = createSymbol('MULTICAST')
export const SAGA_ACTION = createSymbol('SAGA_ACTION')
export const SELF_CANCELLATION = createSymbol('SELF_CANCELLATION')
export const TASK = createSymbol('TASK')
export const TASK_CANCEL = createSymbol('TASK_CANCEL')
export const TERMINATE = createSymbol('TERMINATE')

export const SAGA_LOCATION = createSymbol('LOCATION')
```

../redux-saga/packages/testing-utils/.babelrc.js

```
const { NODE_ENV, BABEL_ENV } = process.env

const cjs = BABEL_ENV === 'cjs' || NODE_ENV === 'test'
const loose = true

module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        loose,
        modules: false,
        exclude: ['transform-regenerator'],
      },
    ],
  ],
  plugins: [cjs && '@babel/plugin-transform-modules-commonjs', 'babel-plugin-annotate-pure-calls'].filter(Boolean),
}
```

../redux-saga/packages/testing-utils/tests/cloneableGenerator.js

```
import { cloneableGenerator } from '../src'

test('it should allow to "clone" the generator', () => {
  const genFunc = function* (num1, num2) {
    yield num1 * num2
    const num3 = yield
    const add = num1 + num2

    if (num3 > add) {
      yield num3 - add
    } else if (num3 === add) {
      yield 'you win'
    } else {
      yield add - num3
    }
  }

  const cloneableGen = cloneableGenerator(genFunc)(2, 3)
  expect(cloneableGen.next()).toEqual({
    value: 6,
    done: false,
  })
  expect(cloneableGen.next()).toEqual({
    value: undefined,
    done: false,
  })
  const cloneElseIf = cloneableGen.clone()
  const cloneElse = cloneElseIf.clone()
  expect(cloneableGen.next(13)).toEqual({
    value: 8,
    done: false,
  })
  expect(cloneableGen.next()).toEqual({
    value: undefined,
    done: true,
  })
  expect(cloneElseIf.next(5)).toEqual({
    value: 'you win',
    done: false,
  })
  expect(cloneElseIf.next()).toEqual({
    value: undefined,
    done: true,
  })
  expect(cloneElse.next(2)).toEqual({
    value: 3,
    done: false,
  })
  const cloneReturn = cloneElse.clone()
  const cloneThrow = cloneElse.clone()
  expect(cloneElse.next()).toEqual({
    value: undefined,
    done: true,
  })
  expect(cloneReturn.return('toto')).toEqual({
    value: 'toto',
    done: true,
  })
  expect(() => cloneThrow.throw('throws an exception')).toThrow()
})
```

../redux-saga/packages/testing-utils/tests/createMockTask.js

```
import { runSaga } from '@redux-saga/core'
import { fork, cancel, join, race, delay } from 'redux-saga/effects'
import { createMockTask } from '../src'

test('can be passed to the cancel effect without an error', () => {
  function* sagaToRun() {}

  function* rootSaga() {
    const task = yield fork(sagaToRun)
    yield cancel(task)
  }

  const taskMock = createMockTask()
  const generator = rootSaga()
  expect(generator.next().value).toEqual(fork(sagaToRun))
  expect(generator.next(taskMock).value).toEqual(cancel(taskMock))
})

test('can be passed to the join effect without an error', () => {
```

```
  function* sagaToRun() {}

  function* rootSaga() {
    const task = yield fork(sagaToRun)
    yield join(task)
  }

  const taskMock = createMockTask()
  const generator = rootSaga()
  expect(generator.next().value).toEqual(fork(sagaToRun))
  expect(generator.next(taskMock).value).toEqual(join(taskMock))
})

test('warns when using deprecated setRunning method', () => {
  const spy = jest.spyOn(console, 'warn')
  const task = createMockTask()
  task.setRunning(false)
  expect(spy).toHaveBeenCalledWith(expect.stringMatching(/setRunning has been deprecated/))
  spy.mockRestore()
})

test('returns a value from being joined when result is set', (done) => {
  runSaga({}, function* saga() {
    const task = createMockTask()
    task.setResult(42)
    const result = yield join(task)
    expect(result).toBe(42)
    done()
  })
})

test('throws an error from being joined when an error is set', (done) => {
  runSaga({}, function* saga() {
    const task = createMockTask()
    const givenErr = new Error('something wrong')
    task.setError(givenErr)
    try {
      yield join(task)
    } catch (err) {
      expect(err).toBe(givenErr)
      done()
    }
  })
})

test('can be cancelled using the cancel effect', (done) => {
  runSaga({}, function* saga() {
    const task = createMockTask()
    yield cancel(task)
    yield join(task)
    done()
  })
})

test('can be cancelled using the cancel method', (done) => {
  runSaga({}, function* saga() {
    const task = createMockTask()
    task.cancel()
    yield join(task)
    done()
  })
})

test('does not resolve a join effect when result is set after passed to join', (done) => {
  runSaga({}, function* saga() {
    const fakeTask = createMockTask()
    const realTask = yield fork(function* () {
      return yield join(fakeTask)
    })
    // Already joined on the task in background, now setting the result
    fakeTask.setResult(42)
    const result = yield race({
      delay: delay(1),
      join: join(realTask),
    })
    expect(result.delay).toBe(true)
    done()
  })
})

test('is running when created', () => {
  const taskMock = createMockTask()
  expect(taskMock.isRunning()).toBe(true)
  expect(taskMock.result()).toBe(undefined)
  expect(taskMock.error()).toBe(undefined)
})

test('is not running after setting the result', () => {
  const taskMock = createMockTask()
  taskMock.setResult(42)
  expect(taskMock.isRunning()).toBe(false)
  expect(taskMock.isAborted()).toBe(false)
  expect(taskMock.isCancelled()).toBe(false)
  expect(taskMock.result()).toBe(42)
})

test('is not running after setting an error', () => {
  const taskMock = createMockTask()
  const err = new Error('Oh no')
  taskMock.setError(err)
  expect(taskMock.isRunning()).toBe(false)
  expect(taskMock.isAborted()).toBe(true)
  expect(taskMock.isCancelled()).toBe(false)
  expect(taskMock.error()).toBe(err)
})

test('is not running after cancelling', () => {
  const taskMock = createMockTask()
  taskMock.cancel()
  expect(taskMock.isRunning()).toBe(false)
  expect(taskMock.isAborted()).toBe(false)
  expect(taskMock.isCancelled()).toBe(true)
```

```
})

test('throws an error when making invalid state transitions', () => {
  const cancelledTask = createMockTask()
  cancelledTask.cancel()
  const cancelledError = /The task is no longer Running, it is Cancelled/
  expect(() => cancelledTask.setResult(42)).toThrowError(cancelledError)
  expect(() => cancelledTask.setError()).toThrowError(cancelledError)
  expect(() => cancelledTask.cancel()).toThrowError(cancelledError)

  const abortedTask = createMockTask()
  abortedTask.setError(new Error('Bad things'))
  const abortedErrorPattern = /The task is no longer Running, it is Aborted/
  expect(() => abortedTask.setResult(42)).toThrowError(abortedErrorPattern)
  expect(() => abortedTask.setError()).toThrowError(abortedErrorPattern)
  expect(() => abortedTask.cancel()).toThrowError(abortedErrorPattern)

  const completedTask = createMockTask()
  completedTask.setResult(42)
  const completedErrorPattern = /The task is no longer Running, it is Done/
  expect(() => completedTask.setResult(42)).toThrowError(completedErrorPattern)
  expect(() => completedTask.setError()).toThrowError(completedErrorPattern)
  expect(() => completedTask.cancel()).toThrowError(completedErrorPattern)
})
```

../redux-saga/packages/testing-utils/babel-transformer.jest.js

```
const path = require('path')
const { createTransformer } = require('babel-7-jest')

module.exports = createTransformer({
  babelrcRoots: path.resolve(__dirname, '../*'),
})
```

../redux-saga/packages/testing-utils/index.d.ts

```
import { SagaIterator, Task, Saga } from '@redux-saga/types'

/**
 * Takes a generator function (function*) and returns a generator function.
 * All generators instanciated from this function will be cloneable.
 * For testing purpose only.
 *
 * #### Example
 *
 * This is useful when you want to test a different branch of a saga without
 * having to replay the actions that lead to it.
 *
 *    import { cloneableGenerator } from '@redux-saga/testing-utils';
 *
 *    function* oddOrEven() {
 *      // some stuff are done here
 *      yield 1;
 *      yield 2;
 *      yield 3;
 *
 *      const userInput = yield 'enter a number';
 *      if (userInput % 2 === 0) {
 *        yield 'even';
 *      } else {
 *        yield 'odd'
 *      }
 *    }
 *
 *    test('my oddOrEven saga', assert => {
 *      const data = {};
 *      data.gen = cloneableGenerator(oddOrEven)();
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        1,
 *        'it should yield 1'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        2,
 *        'it should yield 2'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        3,
 *        'it should yield 3'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        'enter a number',
 *        'it should ask for a number'
 *      );
 *
 *      assert.test('even number is given', a => {
 *        // we make a clone of the generator before giving the number;
 *        data.clone = data.gen.clone();
 *
 *        a.equal(
 *          data.gen.next(2).value,
 *          'even',
 *          'it should yield "even"'
 *        );
 *
 *        a.equal(
 *          data.gen.next().done,
 *          true,
 *          'it should be done'
 *        );
```

```
 *
 *        a.end();
 *      });
 *
 *      assert.test('odd number is given', a => {
 *
 *        a.equal(
 *          data.clone.next(1).value,
 *          'odd',
 *          'it should yield "odd"'
 *        );
 *
 *        a.equal(
 *          data.clone.next().done,
 *          true,
 *          'it should be done'
 *        );
 *
 *        a.end();
 *      });
 *
 *      assert.end();
 *    });
 */
export function cloneableGenerator<S extends Saga>(saga: S): (...args: Parameters<S>) => SagaIteratorClone

export interface SagaIteratorClone extends SagaIterator {
  clone: () => SagaIteratorClone
}

/**
 * Returns an object that mocks a task.
 * For testing purposes only.
 */
export function createMockTask(): MockTask

export interface MockTask extends Task {
  setRunning(running: boolean): void
  setResult(result: any): void
  setError(error: any): void
}
```

../redux-saga/packages/testing-utils/jest.config.js

```
const lernaAliases = require('lerna-alias').jest()

module.exports = {
  testEnvironment: 'node',
  moduleNameMapper: Object.assign(lernaAliases, {
    '^redux-saga/effects$': lernaAliases['^redux-saga$'].replace(/index\.js$/, 'effects.js'),
    '^@redux-saga/core/effects$': lernaAliases['^@redux-saga/core$'].replace(/index\.js$/, 'effects.js'),
  }),
  transform: {
    '.js$': __dirname + '/babel-transformer.jest.js',
  },
}
```

../redux-saga/packages/testing-utils/rollup.config.js

```
import babel from 'rollup-plugin-babel'
import pkg from './package.json'

const makeExternalPredicate = (externalArr) => {
  if (!externalArr.length) {
    return () => false
  }
  const pattern = new RegExp(`^(${externalArr.join('|')})($|/)`)
  return (id) => pattern.test(id)
}

const deps = Object.keys(pkg.dependencies || {})
const peerDeps = Object.keys(pkg.peerDependencies || {})

const createConfig = ({ output, useESModules = output.format !== 'cjs' }) => ({
  input: 'src/index.js',
  output: {
    exports: 'named',
    ...output,
  },
  external: makeExternalPredicate(deps.concat(peerDeps)),
  plugins: [
    babel({
      exclude: 'node_modules/**',
      babelHelpers: 'runtime',
      plugins: [
        [
          '@babel/plugin-transform-runtime',
          {
            useESModules,
          },
        ],
      ],
    }),
  ],
})

export default [
  createConfig({
    output: {
      file: pkg.module,
      format: 'esm',
    },
  }),
  createConfig({
    output: {
      file: pkg.main,
      format: 'cjs',
```

```
    },
  }),
]
```

../redux-saga/packages/testing-utils/src/index.js

```
import { TASK } from '@redux-saga/symbols'

// Keep in sync with @redux-saga/core/src/internal/task-status
const RUNNING = 0
const CANCELLED = 1
const ABORTED = 2
const DONE = 3

const statusToStringMap = {
  [RUNNING]: 'Running',
  [CANCELLED]: 'Cancelled',
  [ABORTED]: 'Aborted',
  [DONE]: 'Done',
}

export const cloneableGenerator =
  (generatorFunc) =>
  (...args) => {
    const history = []
    const gen = generatorFunc(...args)
    return {
      next: (arg) => {
        history.push(arg)
        return gen.next(arg)
      },
      clone: () => {
        const clonedGen = cloneableGenerator(generatorFunc)(...args)
        history.forEach((arg) => clonedGen.next(arg))
        return clonedGen
      },
      return: (value) => gen.return(value),
      throw: (exception) => gen.throw(exception),
    }
  }

const assertStatusRunning = (status) => {
  if (status !== RUNNING) {
    const str = statusToStringMap[status]
    throw new Error(
      `The task is no longer Running, it is ${str}. You can't change the status of a task once it is no longer running.`,
    )
  }
}

export function createMockTask() {
  let status = RUNNING
  let taskResult
  let taskError

  return {
    [TASK]: true,
    isRunning: () => status === RUNNING,
    isCancelled: () => status === CANCELLED,
    isAborted: () => status === ABORTED,
    result: () => taskResult,
    error: () => taskError,
    cancel: () => {
      assertStatusRunning(status)
      status = CANCELLED
    },
    joiners: [],

    /**
     * @deprecated Use `setResult`, `setError`, or `cancel` to change the
     * running status of the mock task.
     */
    setRunning: () => {
      // eslint-disable-next-line no-console
      console.warn(
        'setRunning has been deprecated. It no longer has any effect when being called. ' +
          'If you were calling setResult or setError followed by setRunning, those methods now change the ' +
          'running status of the task. Simply remove the call to setRunning for the desired behavior.',
      )
    },
    setResult: (r) => {
      assertStatusRunning(status)
      taskResult = r
      status = DONE
    },
    setError: (e) => {
      assertStatusRunning(status)
      taskError = e
      status = ABORTED
    },
  }
}
```

../redux-saga/packages/testing-utils/types/cloneableGenerator.test.ts

```
import { SagaIterator } from 'redux-saga';
import { put } from 'redux-saga/effects';
import { cloneableGenerator } from '@redux-saga/testing-utils';

function testCloneableGenerator() {
  function* testSaga(): SagaIterator {
    yield put({type: 'my-action'});
  }

  const cloneableGen = cloneableGenerator(testSaga)();
  const value = cloneableGen.next().value;
```

```
  const clone = cloneableGen.clone();
  const cloneVal = clone.next().value;
}

function testCloneableGenerator1() {
  function* testSaga(n1: number): SagaIterator {
    yield put({type: 'my-action'});
  }

  // $ExpectError
  cloneableGenerator(testSaga)();

  // $ExpectError
  cloneableGenerator(testSaga)('foo');

  cloneableGenerator(testSaga)(1);
}

function testCloneableGenerator2() {
  function* testSaga(n1: number, n2: number): SagaIterator {
    yield put({type: 'my-action'});
  }
  cloneableGenerator(testSaga)(1, 2);
}

function testCloneableGenerator3() {
  function* testSaga(n1: number, n2: number, n3: number): SagaIterator {
    yield put({type: 'my-action'});
  }

  // $ExpectError
  cloneableGenerator(testSaga)(1, 2);

  cloneableGenerator(testSaga)(1, 2, 3);
}

function testCloneableGenerator4() {
  function* testSaga(
    n1: number,
    n2: number,
    n3: number,
    n4: number,
  ): SagaIterator {
    yield put({type: 'my-action'});
  }
  cloneableGenerator(testSaga)(1, 2, 3, 4);
}

function testCloneableGenerator5() {
  function* testSaga(
    n1: number,
    n2: number,
    n3: number,
    n4: number,
    n5: number,
  ): SagaIterator {
    yield put({type: 'my-action'});
  }
  cloneableGenerator(testSaga)(1, 2, 3, 4, 5);
}

function testCloneableGenerator6() {
  function* testSaga(
    n1: number,
    n2: number,
    n3: number,
    n4: number,
    n5: number,
    n6: number,
  ): SagaIterator {
    yield put({type: 'my-action'});
  }
  cloneableGenerator(testSaga)(1, 2, 3, 4, 5, 6);
}

function testCloneableGenerator6Rest() {
  function* testSaga(
    n1: number,
    n2: number,
    n3: number,
    n4: number,
    n5: number,
    n6: number,
    n7: number,
  ): SagaIterator {
    yield put({type: 'my-action'});
  }
  cloneableGenerator(testSaga)(1, 2, 3, 4, 5, 6, 7);
}
```

../redux-saga/packages/testing-utils/types/index.d.ts

```
// TypeScript Version: 4.2
import { SagaIterator, Task, Saga } from '@redux-saga/types'

/**
 * Takes a generator function (function*) and returns a generator function.
 * All generators instanciated from this function will be cloneable.
 * For testing purpose only.
 *
 * #### Example
 *
 * This is useful when you want to test a different branch of a saga without
 * having to replay the actions that lead to it.
 *
 *    import { cloneableGenerator } from '@redux-saga/testing-utils';
 *
 *    function* oddOrEven() {
 *      // some stuff are done here
```

```
 *      yield 1;
 *      yield 2;
 *      yield 3;
 *
 *      const userInput = yield 'enter a number';
 *      if (userInput % 2 === 0) {
 *        yield 'even';
 *      } else {
 *        yield 'odd'
 *      }
 *    }
 *
 *    test('my oddOrEven saga', assert => {
 *      const data = {};
 *      data.gen = cloneableGenerator(oddOrEven)();
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        1,
 *        'it should yield 1'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        2,
 *        'it should yield 2'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        3,
 *        'it should yield 3'
 *      );
 *
 *      assert.equal(
 *        data.gen.next().value,
 *        'enter a number',
 *        'it should ask for a number'
 *      );
 *
 *      assert.test('even number is given', a => {
 *        // we make a clone of the generator before giving the number;
 *        data.clone = data.gen.clone();
 *
 *        a.equal(
 *          data.gen.next(2).value,
 *          'even',
 *          'it should yield "even"'
 *        );
 *
 *        a.equal(
 *          data.gen.next().done,
 *          true,
 *          'it should be done'
 *        );
 *
 *        a.end();
 *      });
 *
 *      assert.test('odd number is given', a => {
 *
 *        a.equal(
 *          data.clone.next(1).value,
 *          'odd',
 *          'it should yield "odd"'
 *        );
 *
 *        a.equal(
 *          data.clone.next().done,
 *          true,
 *          'it should be done'
 *        );
 *
 *        a.end();
 *      });
 *
 *      assert.end();
 *    });
 */
export function cloneableGenerator<S extends Saga>(saga: S): (...args: Parameters<S>) => SagaIteratorClone

export interface SagaIteratorClone extends SagaIterator {
  clone: () => SagaIteratorClone
}

/**
 * Returns an object that mocks a task.
 * For testing purposes only.
 */
export function createMockTask(): MockTask

export interface MockTask extends Task {
  /**
   * @deprecated Use {@link setResult}, {@link setError}, or {@link cancel} to
   * change the running status of the mock task.
   */
  setRunning(running: boolean): void
  setResult(result: any): void
  setError(error: any): void
}
```

../redux-saga/packages/types/types/index.d.ts

```
// TypeScript Version: 3.2
export interface Action<T extends string = string> {
  type: T
}

export type Saga<Args extends any[] = any[]> = (...args: Args) => IterableIterator<any>
```

```
/**
 * Annotate return type of generators with `SagaIterator` to get strict
 * type-checking of yielded effects.
 */
export type SagaIterator = IterableIterator<StrictEffect>

export type GuardPredicate<G extends T, T = any> = (arg: T) => arg is G

export type ActionType = string | number | symbol

export type Predicate<T> = (arg: T) => boolean

export type StringableActionCreator<A extends Action = Action> = {
  (...args: any[]): A
  toString(): string
}

export type SubPattern<T> = Predicate<T> | StringableActionCreator | ActionType

export type Pattern<T> = SubPattern<T> | SubPattern<T>[]

export type ActionSubPattern<Guard extends Action = Action> =
  | GuardPredicate<Guard, Action>
  | StringableActionCreator<Guard>
  | Predicate<Action>
  | ActionType

export type ActionPattern<Guard extends Action = Action> = ActionSubPattern<Guard> | ActionSubPattern<Guard>[]

export type ActionMatchingPattern<P extends ActionPattern> = P extends ActionSubPattern
  ? ActionMatchingSubPattern<P>
  : P extends ActionSubPattern[] ? ActionMatchingSubPattern<P[number]> : never

export type ActionMatchingSubPattern<P extends ActionSubPattern> = P extends GuardPredicate<infer A, Action>
  ? A
  : P extends StringableActionCreator<infer A> ? A : Action

export type NotUndefined = {} | null

/**
 * Used to implement the buffering strategy for a channel. The Buffer interface
 * defines 3 methods: `isEmpty`, `put` and `take`
 */
export interface Buffer<T> {
  /**
   * Returns true if there are no messages on the buffer. A channel calls this
   * method whenever a new taker is registered
   */
  isEmpty(): boolean
  /**
   * Used to put new message in the buffer. Note the Buffer can choose to not
   * store the message (e.g. a dropping buffer can drop any new message
   * exceeding a given limit)
   */
  put(message: T): void
  /**
   * used to retrieve any buffered message. Note the behavior of this method has
   * to be consistent with `isEmpty`
   */
  take(): T | undefined
  flush(): T[]
}

/**
 * A channel is an object used to send and receive messages between tasks.
 * Messages from senders are queued until an interested receiver request a
 * message, and registered receiver is queued until a message is available.
 *
 * Every channel has an underlying buffer which defines the buffering strategy
 * (fixed size, dropping, sliding)
 *
 * The Channel interface defines 3 methods: `take`, `put` and `close`
 */
export interface Channel<T extends NotUndefined> {
  /**
   * Used to register a taker. The take is resolved using the following rules
   *
   * - If the channel has buffered messages, then `callback` will be invoked
   *   with the next message from the underlying buffer (using `buffer.take()`)
   * - If the channel is closed and there are no buffered messages, then
   *   `callback` is invoked with `END`
   * - Otherwise`callback` will be queued until a message is put into the
   *   channel
   */
  take(cb: (message: T | END) => void): void
  /**
   * Used to put message on the buffer. The put will be handled using the
   * following rules
   *
   * - If the channel is closed, then the put will have no effect.
   * - If there are pending takers, then invoke the oldest taker with the
   *   message.
   * - Otherwise put the message on the underlying buffer
   */
  put(message: T | END): void
  /**
   * Used to extract all buffered messages from the channel. The flush is
   * resolved using the following rules
   *
   * - If the channel is closed and there are no buffered messages, then
   *   `callback` is invoked with `END`
   * - Otherwise `callback` is invoked with all buffered messages.
   */
  flush(cb: (items: T[] | END) => void): void
  /**
   * Closes the channel which means no more puts will be allowed. All pending
   * takers will be invoked with `END`.
   */
  close(): void
}
```

```
export type Effect<T = any> = SimpleEffect<T, any> | CombinatorEffect<T, any>

export type StrictEffect<T = any> = SimpleEffect<T, any> | StrictCombinatorEffect<T>

export interface StrictCombinatorEffect<T> extends CombinatorEffect<T, StrictEffect<T>> {}

export interface SimpleEffect<T, P> {
  '@@redux-saga/IO': true
  combinator: false
  type: T
  payload: P
}

/**
 * `all` / `race` effects
 */
export interface CombinatorEffect<T, E> {
  '@@redux-saga/IO': true
  combinator: true
  type: T
  payload: CombinatorEffectDescriptor<E>
}

export type CombinatorEffectDescriptor<E> = { [key: string]: E } | E[]

export type END = { type: '@@redux-saga/CHANNEL_END' }

/**
 * The Task interface specifies the result of running a Saga using `fork`,
 * `middleware.run` or `runSaga`.
 */
export interface Task<T = any> {
  /**
   * Returns true if the task hasn't yet returned or thrown an error
   */
  isRunning(): boolean
  /**
   * Returns true if the task has been cancelled
   */
  isCancelled(): boolean
  /**
   * Returns task return value. `undefined` if task is still running
   */
  result<R = T>(): R | undefined
  /**
   * Returns task thrown error. `undefined` if task is still running
   */
  error(): any | undefined
  /**
   * Returns a Promise which is either:
   * - resolved with task's return value
   * - rejected with task's thrown error
   */
  toPromise<R = T>(): Promise<R>
  /**
   * Cancels the task (If it is still running)
   */
  cancel(): void
  setContext<C extends object>(props: Partial<C>): void
}
```

../redux-saga/packages/types/types/ts3.6/index.d.ts

```
export interface Action<T extends string = string> {
  type: T
}

export type Saga<Args extends any[] = any[]> = (...args: Args) => Iterator<any>

/**
 * Annotate return type of generators with `SagaIterator` to get strict
 * type-checking of yielded effects.
 */
export type SagaIterator<RT = any> = Iterator<StrictEffect, RT, any>

export type GuardPredicate<G extends T, T = any> = (arg: T) => arg is G

export type ActionType = string | number | symbol

export type Predicate<T> = (arg: T) => boolean

export type StringableActionCreator<A extends Action = Action> = {
  (...args: any[]): A
  toString(): string
}

export type SubPattern<T> = Predicate<T> | StringableActionCreator | ActionType

export type Pattern<T> = SubPattern<T> | SubPattern<T>[]

export type ActionSubPattern<Guard extends Action = Action> =
  | GuardPredicate<Guard, Action>
  | StringableActionCreator<Guard>
  | Predicate<Action>
  | ActionType

export type ActionPattern<Guard extends Action = Action> = ActionSubPattern<Guard> | ActionSubPattern<Guard>[]

export type ActionMatchingPattern<P extends ActionPattern> = P extends ActionSubPattern
  ? ActionMatchingSubPattern<P>
  : P extends ActionSubPattern[] ? ActionMatchingSubPattern<P[number]> : never

export type ActionMatchingSubPattern<P extends ActionSubPattern> = P extends GuardPredicate<infer A, Action>
  ? A
  : P extends StringableActionCreator<infer A> ? A : Action

export type NotUndefined = {} | null

/**
```

```
 * Used to implement the buffering strategy for a channel. The Buffer interface
 * defines 3 methods: `isEmpty`, `put` and `take`
 */
export interface Buffer<T> {
  /**
   * Returns true if there are no messages on the buffer. A channel calls this
   * method whenever a new taker is registered
   */
  isEmpty(): boolean
  /**
   * Used to put new message in the buffer. Note the Buffer can choose to not
   * store the message (e.g. a dropping buffer can drop any new message
   * exceeding a given limit)
   */
  put(message: T): void
  /**
   * used to retrieve any buffered message. Note the behavior of this method has
   * to be consistent with `isEmpty`
   */
  take(): T | undefined
  flush(): T[]
}

/**
 * A channel is an object used to send and receive messages between tasks.
 * Messages from senders are queued until an interested receiver request a
 * message, and registered receiver is queued until a message is available.
 *
 * Every channel has an underlying buffer which defines the buffering strategy
 * (fixed size, dropping, sliding)
 *
 * The Channel interface defines 3 methods: `take`, `put` and `close`
 */
export interface Channel<T extends NotUndefined> {
  /**
   * Used to register a taker. The take is resolved using the following rules
   *
   * - If the channel has buffered messages, then `callback` will be invoked
   *   with the next message from the underlying buffer (using `buffer.take()`)
   * - If the channel is closed and there are no buffered messages, then
   *   `callback` is invoked with `END`
   * - Otherwise`callback` will be queued until a message is put into the
   *   channel
   */
  take(cb: (message: T | END) => void): void
  /**
   * Used to put message on the buffer. The put will be handled using the
   * following rules
   *
   * - If the channel is closed, then the put will have no effect.
   * - If there are pending takers, then invoke the oldest taker with the
   *   message.
   * - Otherwise put the message on the underlying buffer
   */
  put(message: T | END): void
  /**
   * Used to extract all buffered messages from the channel. The flush is
   * resolved using the following rules
   *
   * - If the channel is closed and there are no buffered messages, then
   *   `callback` is invoked with `END`
   * - Otherwise `callback` is invoked with all buffered messages.
   */
  flush(cb: (items: T[] | END) => void): void
  /**
   * Closes the channel which means no more puts will be allowed. All pending
   * takers will be invoked with `END`.
   */
  close(): void
}

export interface Effect<T = any, P = any> {
  '@@redux-saga/IO': true
  combinator: boolean
  type: T
  payload: P
}

export interface SimpleEffect<T, P = any> extends Effect<T, P> {
  combinator: false
}

export type StrictEffect<T = any, P = any> = SimpleEffect<T, P> | StrictCombinatorEffect<T, P>;

/**
 * `all` / `race` effects
 */
export type ArrayCombinatorEffectDescriptor<E = any> = E[]
export type ObjectCombinatorEffectDescriptor<E = any> = {[key: string]: E}
export type CombinatorEffectDescriptor<E = any> =
  | ArrayCombinatorEffectDescriptor<E>
  | ObjectCombinatorEffectDescriptor<E>

export interface CombinatorEffect<T, P> extends Effect<
  T, CombinatorEffectDescriptor<P>
> {
  combinator: true
}

export interface StrictCombinatorEffect<T, P> extends Effect<
  T, CombinatorEffectDescriptor<StrictEffect>
> {
  combinator: true
}

export type END = { type: '@@redux-saga/CHANNEL_END' }

/**
 * The Task interface specifies the result of running a Saga using `fork`,
 * `middleware.run` or `runSaga`.
 */
export interface Task<T = any> {
```

```
  /**
   * Returns true if the task hasn't yet returned or thrown an error
   */
  isRunning(): boolean
  /**
   * Returns true if the task has been cancelled
   */
  isCancelled(): boolean
  /**
   * Returns task return value. `undefined` if task is still running
   */
  result<R = T>(): R | undefined
  /**
   * Returns task thrown error. `undefined` if task is still running
   */
  error(): any | undefined
  /**
   * Returns a Promise which is either:
   * - resolved with task's return value
   * - rejected with task's thrown error
   */
  toPromise<R = T>(): Promise<R>
  /**
   * Cancels the task (If it is still running)
   */
  cancel(): void
  setContext<C extends object>(props: Partial<C>): void
}
```